Contents

[GLOBAL] **11**

[24-06-18] **11**
Sources: . . . 12

[25-06-18] **12**

[26-06-18] **13**
Snippets . . . 13
Rot mats . . . 13
Perspective matrix . . . 14

[27-06-18] english.remove() french.add() **14**

[30-06-18] **15**
[DEPS] LINUX OPENGL WINDOW . . . 15
[LINKING] . . . 15
[BUG][fixed] GL segfault à glGenVertexArrays() . . . 15
[Transformation] . . . 15
[BufferUnit] . . . 16

[01-07-18] **16**
Camera . . . 16
[Shader] . . . 16
[Shader] . . . 17

[02-07-18] **17**
[BUG][fixed] CATCH2 segfault . . . 17
[Bezier] . . . 17
Note pour le turfu . . . 18

[03-07-18] **18**

[boost::gil][deprec] **18**
[PngLoader]&[PixelBuffer] . . . 19
UPDATE: CATCH2 segfault . . . 19
[Texture] . . . 19
[H_] La macro qu'elle est trop bien contrairement à ce que j'ai pu dire avant sur les macros parce que cette macro-là ne fait rien du tout . . . 20
DEBUG & PROFILING OPTIONS . . . 20

[04-07-18] **21**
[Bezier] . . . 21
[Texture] . . . 21

[05-07-18] **21**
[Texture] . . . 21
[Material] . . . 22
Cmd arguments . . . 22
[Texture] Named textures . . . 22
UPDATE[H_] . . . 23
[Texture] Debbuging . . . 23
[TODO] Suite du programme . . . 23

[06-07-18] **23**

[BUG][fixed] Test overlay not displaying **24**
Fix temporaire -> Fix définitif . . . 24
[PostProcessing] . . . 24
[TODO] Suite du programme . . . 24
"Physically based shading" . . . 25

[07-07-18] **25**
[ScreenRenderer]&[Scene] . . . 25
[BUG][fixed] Pbm affichage terrain . . . 26
[HeightMap] & mesh factory . . . 27
Truc bizarre ? . . . 28

[08-07-18] **28**
[Lighting] . . . 28
[Texture][OPT] . . . 28
[Light] & [DirectionalLight] . . . 29
[Game loop] . . . 29
[ANSI Escape] . . . 30
[BUG][fixed] Crash graphique lourdingue . . . 30
UPDATE . . . 30
Idées pour les uniforms de PointLights multiples . . . 30
[TODO] Suite du programme : Going full PBR . . . 31

[09-07-18] **31**
[DEBUG] Notes de session . . . 31
[BUG][fixed] Instabilité shader . . . 32

[10-07-18] **32**
[Lighting] . . . 32
[DEBUG] Memory leaks . . . 32

[11-07-18] **33**
[FrameBuffer] . . . 33
[Texture] & [FrameBuffer] . . . 34

[12-07-18] **34**
[GBuffer] . . . 34
[Defered Shading] . . . 35
[HDR] . . . 35
[sRGB] . . . 36
[Lighting][reference] Point lights coefficients . . . 36
Sources : . . . 36
[TODO] Suite du programme . . . 36

[13-07-18] **36**
Sources : . . . 37
[Zik] . . . 37
[Bloom] . . . 37
[ProcessingStage] . . . 37
[Bright Pass] sorta... . . . 38
[BlurPass] . . . 38

[14-07-18] **38**
[Bloom] . . . 38
[BUG][fixed] . . . 38
Textures bloom de taille puissance de 2 . . . 38
[Command Line Galore] . . . 39
[TODO] Suite du programme . . . 39

Sources : . . . 39

[16-07-18] **40**
[BUG][fixed] Specular reflections . . . 40

[17-07-18] **41**
[Procedural] Génération procédurale de cristaux . . . 41
[FXAA] Fast Approximate Anti-Aliasing, L'algo qui fonctionne, et qui fait pas chier. . . . 41
Sources: . . . 42
Notes . . . 42
[Fog] . . . 42

[18-07-18] **42**
[ERROR][~fixed] G-Buffer with internal format GL_UNSIGNED_BYTE . . . 43
[Deferred] +Forward . . . 43
[TODO] Améliorations . . . 44

[19-07-18] **44**

[Normal Mapping] **44**
Notes . . . 45
[GBuffer] Optimisation : compression des normales . . . 45
[TODO] Améliorations et suite du programme . . . 46
[Command Line Galore] . . . 46
[Quotes] de porc . . . 46
[Lighting] Nouvelle fonction d'atténuation . . . 47

[20-07-18] **47**
[Parallax Mapping] . . . 47
Sources: . . . 47
[HACK] . . . 47
[Optim] . . . 47
Note sur la mesure du temps de rendu : . . . 48
Debug geometry . . . 48
[AABB] Axis Aligned Bounding Boxes . . . 48

[22-07-18] Refactoring **48**
[InputHandler] . . . 49
[Frustum Culling] . . . 49
[Serialize] Flatbuffers . . . 50

[23-07-18] **50**
[Optim] . . . 50
[Scene sorting] . . . 50
[Frustum Culling] . . . 50
[Optim] Light Volumes . . . 51
[Zik] . . . 52

[24-07-18] **52**
[Bloom] . . . 52
Camera . . . 52

[25-07-18] **52**
[Deferred Rendering] Stencil Optimization . . . 53
Sources: . . . 53
[Hue Shift] Juste au cas où . . . 53

[27-07-18] **54**

[Text Rendering] . . . 54
[Light Volumes] . . . 55
[Profiling] . . . 55

[28-07-18] 56

[30-07-18] 57
[PBR] . . . 57

[02-08-18] 57
[Soft Shadows] . . . 58

[05-08-18] 59
[TODO][x] Faire 2 shaders séparés pour les lumières directionnelle et ponctuelles. . . . 59

[17-08-18] 59
[XML] Format de map . . . 60
[Chunk] . . . 60
[Parsing] . . . 63
[Color] HSL . . . 63
Refactor . . . 65
[Singletonization] . . . 65
[BufferUnit] Leur place est-elle bien dans un renderer ? . . . 65
[Simplifications] . . . 66
[Chunk System] Préparation . . . 66

[31-08-18] Des pommes, des bananes, des figues 67
[Chunk] . . . 67
[Motion] . . . 67
Grosse parenthèse . . . 68
[XML] . . . 70
[Terrain] . . . 71
Erosion . . . 73
[Material] . . . 73
[BUG][fixed] Transparent models VBO misindexing . . . 73
[Hermite Splines] . . . 73
[Daylight System] . . . 74
[BUG][fixed] Fresnel black disk artifact . . . 75
[Optim] Gaussian Spherical Fresnel approximation . . . 75
[Optim] LoD enabled parallax mapping . . . 75

[02-09-18] Ero-Zi0n 75
[Terrain] Scale : Une échelle pour aller au ciel . . . 75
Erosion Hydro-Erosion : Un algo bien classe . . . 76

[05/06-09-18] Doc à écrire 77

[06-09-18] Better everything 77
[Shader] Améliorations . . . 77
Fin de l'invasion des send_uniform_X() . . . 77
#include "yo_mamma.glsl" . . . 77
[InputHandler] True to its name . . . 77
Enregistrement d'actions . . . 77
Vrai debouncing . . . 78
Parsing XML . . . 78
[DebugOverlayRenderer] Moins ad hoc . . . 79
[Motion] Classe ConstantRotator . . . 79
[Scene] . . . 79

Simplification de l'interface . . . 80
Fin du hard-coding . . . 80
[TerrainPatch] . . . 80
[SSAO] Depuis le temps que j'en parle . . . 80
SSAO world space : Théorie . . . 80
Implémentation . . . 81
Sources : . . . 82
En pratique . . . 82
[SSS] Application détournée de la SSAO . . . 83
Sources : . . . 83

[07/08-09-18] Doc à écrire / Features **83**

[08-09-18] De l'utile, de l'agréable **83**
[Logger] Debugger avec style . . . 83
[Shader] Un système de defines trans-shader semi-automatisé . . . 84
[Stats] Amélioration de la FIFO de temps de rendu . . . 84
[Variance Shadow Mapping] Parce que PCFSM date de 1987 . . . 85

[25-09-18] Doc à écrire **85**

[02-10-18] Ca ressemble de loin à un niveau **85**
[Pipeline] Roi du pétrole . . . 85
Better main() . . . 85
[Chunks] Un gros morceau . . . 86
Refactor Méthode . . . 86
Refactor Scene . . . 86

[07-10-18] Doc à écrire **89**

[12-10-18] Spline l'Ancien **90**

[TreeGenerator] Comment faire un arbre avec des splines **90**
[Spline Skinning] Tentacules . . . 90
[Spline Tree Skinning] . . . 92
[Paramétrisation] . . . 92

[14/15-10-18] TODO **93**

[15-10-18] Doc à écrire **94**

[18-10-18] Bosser sous speed... **94**
[Shadow Mapping] Orthographic frustum tight fit optimization . . . 94
Dans la série "perte de temps débile" . . . 95

[19-10-18] **96**
[Point Light Attenuation] . . . 96

[TODO] **96**

[20-10-18] **96**
[Seamless Terrain] . . . 96

[22-10-18] Petite session Mesh et Shadow Mapping **97**
Refactor Mesh . . . 97
[Shadow Mapping] Optimisations . . . 97

[24-10-18] **97**
Reconstruction de la position depuis le depth buffer . . . 97

[BUG][fixed] State Leak (screen texture) . . . 98

[27-10-18] UI, UI, UIIIIII **98**
Immediate mode . . . 98
Intégration . . . 98
Plan d'action . . . 99
Remarque importante sur l'initialisation des états du GUI . . . 100
Bug ImGui : . . . 100
[Apitrace] Magie noire . . . 100
Sources: . . . 101

[29-10-18] Opti-zonions **101**
[Améliorations] . . . 101
Sources: . . . 103

[30-10-18] **103**
[Bug] HeightmapGenerator fail in target RelWithDebInfo . . . 103
[Améliorations] . . . 103

[01-11-18] **103**
[Améliorations] . . . 103
Sources : . . . 103

[02-11-18] C++ Black Magic **103**
Utilisation pratique . . . 106
SFINAE et std::enable_if . . . 106
Tag dispatching . . . 108
Sources : . . . 108

[04-11-18] **109**
[Position reconstruction] . . . 109
Sources : . . . 110

[05-11-18] **110**
[Position reconstruction] . . . 110

[06-11-18] **110**
[BUG][fixed] Flickering Black Rectangle . . . 110
[BUG][fixed] Anti-reflections . . . 111
[GUI] . . . 111

[07-11-18] **111**
Git . . . 111
Sphères et domes texturés . . . 112

[12-11-18] Refactoring du système d'assets **112**
[TODO][x] Le jeu DOIT être exécuté depuis le dossier build sinon ça ne fonctionne pas. C'est dû aux nombreux paths hardcodés. Corriger ça. . . . 113
Sources: . . . 113

[15-11-18] Better terrain **113**

[21-11-18] Après le repos, la guerre. **113**

[23-11-18] Better logger **114**

[26-11-18] L'important c'est de coder **114**
Better logger cont'd . . . 114
[filesystem] Cauchemar cannabinique . . . 115

Self-path bootstrap . . . 116
START_LEVEL . . . 116

[27-11-18] 117
Exceptions . . . 117

[28-11-18] API culture 117

Messaging 117

[API] Ayéééé ! 118

[30-11-18] Pybind11 119
Solution . . . 120
Script directory . . . 121
Sources : . . . 121

[01-12-18] 122
ECS API . . . 122

[09-12-18] Cotire 122

[12-12-18] Post-processing 123
[PingPongBuffer] . . . 124
[SSAO] Ajout d'un paramètre de biais vectoriel de la distribution d'échantillons . . . 124
[SSDO] Expérimentation avec l'occlusion directionnelle . . . 124

[15-12-18] 125
Toolchain . . . 125
On the fly Gaussian kernels . . . 125

[17-12-18] 126

[19-12-18] 126
Custom cursor . . . 126
Ray Casting . . . 127

[20-12-18] 127
Debug display requests: segments . . . 127

[21-12-18] Pas de repos pour les braves 130

Ray casting Screen -> NDC -> World 130
Ray/AABB intersection . . . 131
Sources : . . . 131

[23-12-18] 131
Ray/OBB intersection . . . 131
GLEW rant . . . 132

[28-12-18] RK integrator 132
Sources : . . . 134

[29-12-18] 134
Sources : . . . 136

[30-12-18] Intern strings & H_ macro 136

[01-01-19] 138

[02-01-19] Du gros Octree qui tache **138**
Sources : . . . 139

[05-01-19] **139**
REFACTOR Bounding boxes . . . 139
Octree . . . 139
Insertion et propagation . . . 139
Suppression et merge . . . 140
Query, traversal, visit . . . 141
Sources : . . . 141

[06-01-19] Binary decision trees FUCK YEAH **142**
Iterative subdivision . . . 142

[12-01-19] **143**
REFACTOR Game systems . . . 143
Consume events . . . 143

[13-01-19] **144**
Collision traits . . . 144

[15-01-19] **145**
Mesh instance . . . 145
Mesh caching . . . 145

[22-01-19] **146**
Sources : . . . 147

[23-01-19] **147**
[TODO][X] Ecrire *EntityFactory* qui permet de construire une entité depuis une description (XML). . . . 147

[08-02-19] **147**
hash litteral . . . 147
SoundSystem . . . 148
InitializerSystem . . . 149
EditorTweaksInitializer . . . 149
Game object factory . . . 149
Mesh caching & instances . . . 149
Application maze . . . 150
Erosion . . . 150
Sources : . . . 150

[12-02-19] **150**
Distance-enabled parallax mapping . . . 150

[17-02-19] **150**
zipios linking . . . 150

[21-02-19] **151**
Archives . . . 151
Better ObjLoader . . . 152

[24-02-19] **152**
Cubemap & Skybox . . . 152
Drawing . . . 153
Merdique (TODO) . . . 154
Sources : . . . 154

[25-02-19] **154**
A propos de SIMD . . . 154
Batch image conversion . . . 155

[27-02-19] **155**
Splat mapping . . . 155

[27-02-19] **156**
Splat mapping . . . 156
[BUG] Octree non robuste . . . 157
Réparation de internstr . . . 157

[01-03-19] **157**
Material Editor . . . 157

[05-03-19] Material Editor **158**
Utilisation d'un modèle proxy pour le tri d'une liste . . . 158
Gestion de la sélection dans une QListView . . . 159
Edition custom . . . 159
Compilation ad hoc des textures . . . 161
Menus contextuels . . . 161
Barres d'outils . . . 162
Gestion des ressources . . . 162
Installation d'un event filter . . . 163
Gestion du drag and drop . . . 163

[08-03-19] **163**
Batch convert images . . . 163

[09-03-19] **164**
Stretch bitch . . . 164
Locale . . . 164
Petite étoile . . . 165

[12-03-19] **166**
[BUILD] . . . 166
Image Processing . . . 167
Scrollable . . . 167
Icons sur Ubuntu/Gnome . . . 168

[15-03-19] **169**
Taille du fb . . . 169
Comment faire un widget OpenGL . . . 170
Orientation de départ de la caméra . . . 171
A faire . . . 171
Sources : . . . 172

[17-03-19] **172**
Valgrind . . . 172
const-correctness . . . 172
Sources : . . . 173

[21-03-19] **173**
Camera . . . 173
Sources : . . . 174

[24-03-19] **174**
LinearPipeline . . . 174

[26-04-19] **175**
Refactor . . . 175

[28-04-19] **175**
SSR: Screen Space Reflections . . . 175
Sources : . . . 177
Framebuffer Peek enfin utile . . . 177
Shader line numbers . . . 178
Shader hot swap . . . 179
Always-on-top . . . 179

[01-05-19] Fête du travail, ça tombe bien, y en a beaucoup **180**
Plan de route pour le moteur d'animation . . . 180

[04-05-19] **181**
XMLSkeletonExporter . . . 181
BinaryMeshExporter et *WeshLoader* . . . 181

[05-05-19] **183**
Notes sur Assimp . . . 183
Sources : . . . 183

[07-05-19] **184**
Textures : MEGA refactor . . . 184

[10-05-19] **184**
Textures : MEGA refactor -> suite et fin . . . 184
Sources : . . . 185
WatFiles . . . 185
Améliorations possibles . . . 186

[11-05-19] **186**
Watfile support refactor . . . 186
BUGS . . . 186

[13-05-19] **186**
Sources : . . . 187

[20-05-19] **187**
Grosse optimisation . . . 187

[21-05-19] **188**
GPU timer query . . . 188
Backface depth-buffer enabled SSR . . . 189
Sources : . . . 190

[25-05-19] **190**
[failed] Bloom pass optimization . . . 190

[27-05-19] **190**
Multi-line string . . . 190

[03-06-19] **190**
Git . . . 190
Remove a git submodule . . . 190
Update all submodules to latest commit from their remote . . . 190

[04-06-19] **190**

[10-06-19] **191**
Vertex Buffer Layout 191

Elegant thread-safe singleton **192**

[GLOBAL]

Pour convertir ces notes en pdf :

```
>> pandoc --latex-engine=xelatex NOTES.md -s -o notes.pdf
```

Avec une belle mise en page :

```
>> pandoc --latex-engine=xelatex --listings -H listings_setup.tex --toc -V
   geometry:"left=2cm, top=2cm, right=2cm, bottom=2cm" -V fontsize=10pt NOTES.md
   -o notes.pdf
```

Et le contenu de listings_setup.tex :

```
\usepackage{xcolor}

\lstset{
    basicstyle=\ttfamily,
    numbers=left,
    numberstyle=\footnotesize,
    stepnumber=2,
    numbersep=5pt,
    backgroundcolor=\color{black!10},
    showspaces=false,
    showstringspaces=false,
    showtabs=false,
    tabsize=2,
    captionpos=b,
    breaklines=true,
    breakatwhitespace=true,
    breakautoindent=true,
    linewidth=\textwidth
}
```

XeTeX est utilisé car il gère nativement l'Unicode. On obtient le package comme suit :

```
>> sudo apt-get install texlive-xetex
```

[24-06-18]

checked Quaternion::get_rotation_matrix() **100 fucking times**. y-axis rotations still *transposed* while they shouldn't be.

```
qy(vec3(0.0, 1.0, 0.0), 45.0);
qy.get_rotation_matrix();
```

```
/ 0.707107     0  *0.707107*  0 \
| 0            1  0           0 |
| *-0.707107*  0  0.707107    0 |
\ 0            0  0           1 /
```

should be

```
init_rotation_euler(Ry, 0, TORADIANS(45.0), 0);
```

```
/ 0.707107  0  -0.707107  0 \
| 0         1  0          0 |
| 0.707107  0  0.707107   0 |
\ 0         0  0          1 /
```

WTF bruh?! Computations are correct,

Sources:

```
[1] wiki, matlab code for quat2rotm
[2] http://www.euclideanspace.com/maths/geometry/rotations/conversions/
quaternionToMatrix/
```

My code gives good rotation matrices from quats compared to what matlab outputs for the same quats. => wrong axis/angle initialization ? (which also seems mathematically rigorous). Same result using Tait-Bryan angles initialization. Damn, nigga.

Matlab code from eul2quat.m:

```
c = cos(eul/2);
s = sin(eul/2);

% The parsed sequence will be in all upper-case letters and validated
switch seq
    case 'ZYX'
        % Construct quaternion
        q = [c(:,1).*c(:,2).*c(:,3)+s(:,1).*s(:,2).*s(:,3), ...
            c(:,1).*c(:,2).*s(:,3)-s(:,1).*s(:,2).*c(:,3), ...
            c(:,1).*s(:,2).*c(:,3)+s(:,1).*c(:,2).*s(:,3), ...
            s(:,1).*c(:,2).*c(:,3)-c(:,1).*s(:,2).*s(:,3)];

    case 'ZYZ'
        % Construct quaternion
        q = [c(:,1).*c(:,2).*c(:,3)-s(:,1).*c(:,2).*s(:,3), ...
            c(:,1).*s(:,2).*s(:,3)-s(:,1).*s(:,2).*c(:,3), ...
            c(:,1).*s(:,2).*c(:,3)+s(:,1).*s(:,2).*s(:,3), ...
            s(:,1).*c(:,2).*c(:,3)+c(:,1).*c(:,2).*s(:,3)];
```

Quaternion::init_tait_bryan() was *corrected* accordingly. Still, we get the flipped y-axis rotation.

OK!! Seems matlab reverses the order of euler angles compared to what I do.

[25-06-18]

Made a quat unit test file using Catch 2 instead of gTest.

Adv: * No need to link against a lib. * Test app main is declared using a #define. * All test code can be organized in other files. Test app is long to compile at first but when the .o is generated, all successive modifications to test implementations take no time at all to compile. * Uses C++ expressions inside a REQUIRE macro, instead of several specialized EXPECT_x/ASSERT_x macros like in gTest. * Generated app is highly parameterized (launch bin/test_math –help)

Disadv: * No mocks.

[26-06-18]

All math stuff is unit tested, except:

```
[x] init_rotation_euler
[x] init_look_at
[x] init_perspective
[x] init_ortho
```

Catch2: New file with this content **only**:

```
    #define CATCH_CONFIG_MAIN
    #include <catch2/catch.hpp>
```

Each test entity in a separate cpp file with the include only (*no define*).

Snippets

tc_ [TAB] -> new Catch2 TEST_CASE() rq_ [TAB] -> REQUIRE();

Rot mats

cf. Notes 24-06

```
    eulx = [0 0 pi/3];
    euly = [0 pi/3 0];
    eulz = [pi/3 0 0];
    rotmx = eul2rotm(eulx)
    rotmy = eul2rotm(euly)
    rotmz = eul2rotm(eulz)
```

```
rotmx =
    1.0000         0         0
         0    0.5000   -0.8660
         0    0.8660    0.5000

rotmy =
     0.5000         0    *0.8660*
          0    1.0000          0
   *-0.8660*        0     0.5000

rotmz =
    0.5000   -0.8660         0
    0.8660    0.5000         0
         0         0    1.0000
```

So there *is* a - sign at rotmy[2]. -> init_rotation_euler() signature corrected accordingly:

```
init_rotation_euler(mat4, Z, Y, X)
```

quats are constructed from Euler angles with the **ZYX convention** too. *DAMMIT I had this all wrong in WEngine, no wonder the cam class was broken as hell…*

Perspective matrix

World (eye) coords are right-handed, but normalized device coordinates (*NDC*) are left-handed for OpenGL and right-handed for DirectX. -> Added a bool leftHanded (default true) to init_perspective just in case (multiply xScale by -1 if right-handed, 1 if left-handed).

[27-06-18] english.remove() french.add()

Je bosse sur l'idée d'une transition smooth ortho -> perspective et inversement.

Ici: https://forum.unity.com/threads/smooth-transition-between-perspective-and-orthographic-modes.32765/ on prétend qu'une *lerp* fait le travail. J'ai mesuré auparavant sous Matlab l'"explosivité" de l'interpolation, à savoir: * le déterminant de l'interpolation * le déterminant de la matrice Sigma d'une décomposition en valeur singulière (le même que précédemment en valeur absolu, Sigma absorbe tout le rescaling). * l'évolution des valeurs propres * tout ceci dans le sens ortho->persp et persp->ortho J'ai un comportement légèrement divergent à l'approche d'ortho (ce qui j'imagine, devait être attendu).

Voilà du code pour Unity, mentionné dans le lien prec.

```
using UnityEngine;
using System.Collections;

[RequireComponent (typeof(Camera))]
public class MatrixBlender : MonoBehaviour
{
    public static Matrix4x4 MatrixLerp(Matrix4x4 from, Matrix4x4 to, float time)
    {
        Matrix4x4 ret = new Matrix4x4();
        for (int i = 0; i < 16; i++)
            ret[i] = Mathf.Lerp(from[i], to[i], time);
        return ret;
    }

    private IEnumerator LerpFromTo(Matrix4x4 src, Matrix4x4 dest, float duration)
    {
        float startTime = Time.time;
        while (Time.time - startTime < duration)
        {
            camera.projectionMatrix = MatrixLerp(src, dest, (Time.time -
               startTime) / duration);
            yield return 1;
        }
        camera.projectionMatrix = dest;
    }

    public Coroutine BlendToMatrix(Matrix4x4 targetMatrix, float duration)
    {
        StopAllCoroutines();
        return StartCoroutine(LerpFromTo(camera.projectionMatrix, targetMatrix,
           duration));
    }
}
```

Suite du programme:

```
[x] Rendre les matrices lerpables.
[ ] Ecrire un début de moteur 3D qui affiche une salle simple en vue ortho.
```

```
    [ ] On doit pouvoir afficher des primitives (cube, cylindre, sphère, systèmes
       d'axes, plans)
    [x] et des modèles.
        [ ] Implémenter un model loader.
[ ] Foutre un perso dans cette salle.
    [ ] Implémenter des contrôles basiques.
[x] Implémenter des courbes lisses (Bézier) 3D
    [ ] les rendre affichables
    [ ] et utilisables comme trajectoire de cam
    [ ] animer la cam de la position traveling courante à la position du player
    [x] lerp la projection entre ortho et perspective
        [ ] enjoy your covfefe
```

[30-06-18]

J'ai unitTest et amélioré les classes suivantes de WEngine dans l'idée de les réut dans WCore:

- Listener / Informer / WData -> core messaging
- WComponent / WEntity -> component oriented entity system
- Vertex types / Mesh

[x] L'objectif pour l'instant est de construire une app simple pouvant afficher un modèle cubique.

[DEPS] LINUX OPENGL WINDOW

```
>> sudo apt-get install cmake make g++ libx11-dev libxi-dev libgl1-mesa-dev
libglu1-mesa-dev libxrandr-dev libxext-dev libxcursor-dev libxinerama-dev
libxi-dev libglew-dev libglfw3-dev
```

[**LINKING**]

On doit linker avec **-lm -lGL -lglfw -lGLEW**

[BUG][fixed] GL segfault à glGenVertexArrays()

Dans le cas où on utilise libGLEW et OpenGL core comme ici, il est capital de modifier la globale glewExperimental = GL_TRUE; *avant* l'appel à glewInit(), sinon on peut très bien segfault dès le premier appel à glGenVertexArrays() (en tout cas, c'est le cas sur ma bécane). L'idée est que le mécanisme principal de GLEW pour récupérer les extensions ne récupère *pas toutes* les extensions…

[Transformation]

J'ai formé une classe *Transformation* qui regroupe un vec3 position, un quaternion pour l'orientation, et un facteur d'échelle (default 1.0f). Cette classe représente une transformation euclidienne appliquée à un modèle. Des fonctions translate et rotate permettent d'ajuster la transformation courante d'un modèle. Un membre get_model_matrix() calcule la matrice monde depuis les trois transformations que l'objet regroupe. La classe *Transformation* est unit testée **à une exception près: rotate_around** qui fait **MEGA** chier. Code actuel:

```
    quat q1(vec4(position_-point, 0.0));
    quat q2(axis, angle);
    quat rot_dist(q2.get_conjugate()*(q1*q2));
    position_ = point+rot_dist.get_as_vec().xyz();
```

Une classe *Model* regroupe pour l'instant un Mesh* et une *Transformation* (plus tard, +Material*). *Model* possède des membres inline rotate et translate qui agissent sur sa *Transformation*. Un autre membre inline get_model_matrix() récupère et transmet la matrice modèle de la *Transformation*.

[BufferUnit]

Une classe *BufferUnit* contient et initialise un VBO et un IBO, et peut absorber un Mesh via la méthode submit() et envoyer ses gros buffers vers OpenGL via une fonction upload(). J'ai choisi la même stratégie que pour WEngine: On enfourne plein de meshs *de même nature* dans un seul gros buffer. Mais dans WCore le VAO est découplé du BufferUnit qui n'est plus qu'un VBO indexé. L'idée est que je peux avoir plusieurs VBO semblables dans un Renderer et un seul VAO associé au Renderer. Comme ça on bind le VAO à l'appel de Renderer.draw() et on dessine les multiples VBOs avec le même interfaçage d'attributs. -> Edit : C'est bien sûr n'importe quoi, à ce stade je suis inconscient de mon erreur. BufferUnit est friend de Mesh, donc il peut appeler la méthode private Mesh::set_buffer_offset(). Ainsi l'affectation de l'offset se fait en background dans la méthode BufferUnit::submit() et on n'a pas à se casser le cul avec en dehors de ces deux classes.

J'ai une classe GLContext qui permet d'ouvrir une fenêtre avec un render target OpenGL (via glfw) et possède une méthode main_loop() qui exécute un callback _setup() avant la boucle do-while et un callback _loop pendant la boucle. Les callbacks sont des std::function que j'initialise pour l'instant avec des lambdas &.

[01-07-18]

Camera

Classe *Camera* "terminée". Elle contient deux Transformation : key_trans_ et trans_. key_trans_ est updatée à chaque appel d'une fonction update (position / orientation). trans_ est updatée à chaque appel de get_view_matrix() :

```
    trans_ *= key_trans_;
    key_trans_.set_identity();
    return trans_.get_model_matrix();
```

La classe contient aussi un math::Frustum et une matrice mat4 de projection. On a pour l'instant une interface assez minimaliste qui permet de la contrôler, de choisir la projection et de récupérer les matrices *View* et *Projection*. Mais le truc cool… C'est que l'on peut aussi choisir une projection hybride entre orthographique et perspective (lerp des deux). Testé avec succès !!

[Shader]

J'ai pour l'instant une classe *Shader* fonctionnelle mais définie juste avant main() (en train d'être prototypée). J'ai bien entendu récupéré des bouts de WEngine pour cette classe. J'ai récupéré les shaders vertex et fragment de Gebobola pour implémenter un per-vertex Gouraud shading vite tef. Bah, ça fonctionne.

Lors du test de la caméra en revanche, je me suis rendu compte d'un problème dans la multiplication de quats qui très étonnamment m'avait échappé (car je suis *on ne peut plus* sérieux avec le unit testing bien sûr…). Encore une permutation circulaire des indices due à des conflits de conventions XYZW / WXYZ. * Donc déjà faudrait revisiter l'operator/ qui doit aussi être écrit de manière frivole. * Et revenir sur le problème d'hier avec le unit test du rotate_around() (qui si ça se trouve doit fonctionner tout seul maintenant).

J'ai fait une "scène" test avec un cube non tex qui tourne en Gouraud shading. J'ai aussi testé la transition ortho->persp et inverse en me servant de cette scène. Ben c'est un bel effet seamless !

[Shader]

- Nous avons maintenant une classe *Shader* toute propre.
- Geometry shader opérationnel, j'ai rendu vie à ce bon vieux bout de code qui me permettait d'afficher le wireframe dans WEngine. Plutôt que d'activer/désactiver cette fonctionnalité en tout ou rien, j'ai paramétré le blend du wireframe avec un float $\in [0, 1]$.

[02-07-18]

[BUG][fixed] CATCH2 segfault

```
>>../bin/test_engine_3d
```

crash comme un enculé. Bug de catch2 ? Dès qu'on inclue la source shader.cpp l'appli de test segfault. C'est une belle merde.

Assert perso:

```
#ifndef NDEBUG
#   define ASSERT(condition, message) \
    do { \
        if (! (condition)) { \
            std::cerr << "Assertion `" #condition "` failed in " << __FILE__ \
                      << " line " << __LINE__ << ": " << message << std::endl; \
            std::terminate(); \
        } \
    } while (false)
#else
#   define ASSERT(condition, message) do { } while (false)
#endif
```

This will define the ASSERT macro only if the no-debug macro NDEBUG isn't defined.

Then you'd use it like this:

```
ASSERT((0 < x) && (x < 10), "x was " << x);
```

Which is a bit simpler than your usage since you don't need to stringify "x was" and x explicitly, this is done implicitly by the macro.

[Bezier]

J'ai codé une classe *Bezier* qui comme son nom l'indique, calcule des interpolations type courbe de Bézier d'ordre n (n+1 pts de contrôle). L'article wiki https://en.wikipedia.org/wiki/B%C3%A9zier_curve parle de l'existence d'une forme polynomiale en puissance du paramètre t:

```
\vec{B}(t) = \sum_{j=0}^{n} t^j \vec{C_j}
```

Les coefficients C_j sont des vecteurs calculable grâce à un réarrangement des termes des polynomes de Bernstein:

```
\vec{C_j} = \prod_{m=0}^{j-1}(n-m) * \sum_{i=0}^{j}
   \frac{(-1)^{i+j}\vec{P_i}}{i!(j-i)!}
```

L'idée est qu'une fois qu'on a les points de contrôle (à l'initialisation de l'objet si possible), on calcule les coeffs en prévision d'évaluations futures, puis chaque évaluation consiste en une bête somme vectorielle pondérée des coeffs (plus rapide qu'un calcul depuis la forme de Bernstein). Naturellement, dès qu'un point de contrôle est modifié, les coefficients sont recalculés. En toute modestie, le code est optim à mort, ça ne devrait pas poser problème.

```
    assert((control_.size() <= nfact) && "Factorials not defined this far.");

    coeffs_.resize(control_.size());
    float prod = 1.0f;
    for (int jj = 0; jj < control_.size(); ++jj)
    {
        if(jj>0)
            prod *= (order()-jj+1);

        vec3 sum(0.0f);
        for (int ii = 0; ii <= jj; ++ii)
            sum += control_[ii] * (((ii+jj)%2)?-1.0f:1.0f) /
               (factorial[ii]*factorial[jj-ii]);
                                  // == pow(-1.0f,ii+jj)
        coeffs_[jj] = prod * sum;
    }
```

J'ai fait suivre une de ces courbes au cube de la scène. Ben ça marche.

Note pour le turfu

g++5.3 va joyeusement compiler

```
    for(aaa : Collection){}
```

Mais pas g++7 qui lui va gueuler à juste titre que aaa n'est pas déclarée. **auto auto auto**

```
    for(auto aaa : Collection){}
```

[03-07-18]

[boost::gil][deprec]

Charger des png… Je compte utiliser boost::gil plutôt que libpng. Boost gil obtensible comme suit:

```
>> git clone https://github.com/boostorg/gil.git
```

Il y a une **dep** avec libtiff:

```
>> sudo apt-get install libtiff5-dev
```

Là, j'essaye de compiler, avec make, ça compile mais ça ne link pas une appli test à cause de libtiff (peut-être qu'il attend une version différente), du coup make n'aboutit pas, et par flemme j'ai juste copié les headers en pensant que ça serait suffisant (comme souvent avec boost):

```
>> sudo cp -R include/boost/* /usr/include/boost/
```

veiller à avoir une version à jour de cmake (dl depuis https://cmake.org/download/) et décompresser puis

```
>> ./configure
>> make
>> sudo make install
```

boost doit bien entendu être à jour aussi. dl depuis https://www.boost.org/

```
>> tar -jxvf boost_1_67_0.tar.bz2
>> cd boost_1_67_0/
>> ./bootstrap.sh
>> ./b2
```

là ça compile tout le bordel (environ 15min). Pour installer, j'ai backup ma vieille version de boost au cas où, puis copié le dossier boost:

```
>> sudo mv /usr/include/boost/ /usr/include/boost_old
>> sudo cp -R ./boost /usr/include/boost
```

Pour se servir de la lib, c'est full include:

```
#include <boost/gil.hpp>
```

Bien entendu, rien de tout ça ne fonctionne au final... Je veux bien me casser le cul à faire fonctionner boost::gil, mais une telle usine à gaz non triviale à setup dont on n'utilise qu'une partie infime tombe dans la catégorie des dépendences à la con.

-> libpng Et là tout fonctionne nickel. Penser à link avec libpng:

```
>> g++ ..... -lpng
```

Je suis ce très bon tuto pour l'implémentation du loader: http://www.piko3d.net/tutorials/libpng-tutorial-loading-png-files-from-streams/

[PngLoader]&[PixelBuffer]

J'ai ma classe *PngLoader*. C'est un singleton avec une méthode load_png() qui prend un path en argument. Cette classe crée (new) un objet *PixelBuffer* et l'initialise avec les données du header png. Le PixelBuffer est ensuite rempli des données de l'image et un pointeur est livré à l'appelant qui est **responsable de sa destruction**.

PixelBuffer possède un constructeur dont le dernier argument est un foncteur d'initialisation. Le code qui crée l'instance peut pousser son code d'initialisation dans un lambda qui est passé au constructeur de PixelBuffer, comme je le fais dans pixel_loader.cpp:

```
    px_buf = new PixelBuffer(imgWidth, imgHeight, bitDepth, channels,
    [&](unsigned char** pp_rows)
    {
        // Read image to pixel buffer
        png_read_image(p_png, pp_rows);
    });
```

[?] Une possible future classe ResourceLoader pourrait cacher les PixelBuffers, et retourner un pointeur vers une instance déjà existante quand on veut à nouveau charger un png déjà en mémoire. Tout ça se ferait à travers l'appel univoque à ResourceLoader::get_png(filename).

UPDATE: CATCH2 segfault

Il suffit de ne pas inclure les cpp qui ont une dep à GL (shader, texture).

[Texture]

Classe *Texture* fonctionnelle (encore un emprunt à WEngine + quelques améliorations). La scène test affiche maintenant un cube texturé. J'ai bien dû faire un one shot, mais je m'en suis rendu compte après 4h de debug/coups de gueule. Le cube était blanc… Voici ce qui se trouvait dans la liste d'init du constructeur de TextureInternal:

```
#ifdef PROFILING_SET_2x2_TEXTURE
    width_(width),
    height_(height),
#else
    width_(2),
    height_(2),
```

Bah ouais, ça réduisait la taille de la texture à 2x2 px, dans une zone blanche… Ce code sert à réduire la taille de toutes les textures à 2x2 pour faire du profiling (idée de ThinMatrix que j'ai reprise dans WEngine). J'ai juste eu à inverser les couples d'instructions. Quand on dit que **les macros c'est CACA**. Bon, il a aussi fallu que je refasse les coords UV dans le stub mesh factory qui produit le cube. *Model* possède maintenant un _Mesh*_ et le *BufferUnit* est maintenant un *BufferUnit*.

Texture possède un std::shared_ptr sur un objet *TextureInternal* qui fait le gros de l'initialisation et qui conserve les données utiles à OpenGL. Lors de la construction d'une *Texture*, si c'est le constructeur qui prend un filepath en argument qui est appelé, le filepath est sauvegardé sous forme de hash (ou tel quel si **PRESERVE_STRS** est défini, voir [H_]) lors du premier accès à la ressource, un nouvel objet *TextureInternal* est créé et initialisé à travers std::make_shared(Args&&…) et le std::shared_ptr résultant est associé au hash du filepath dans une unordered_map. Ainsi lors de futurs accès à la même ressource, je peux simplement fabriquer un shared_ptr à la volée au lieu de générer une nouvelle copie de la texture. L'idée est encore de ThinMatrix.

[H_] La macro qu'elle est trop bien contrairement à ce que j'ai pu dire avant sur les macros parce que cette macro-là ne fait rien du tout

Dans utils.h j'ai une fonction constexpr récursive pour le calcul des hash de strings char* compile-time. H_() est alors une fonction qui retourne un hash et hash_t est un type numérique. Si **PRESERVE_STRS** est défini, H_() est en revanche une macro qui ne fait rien du tout et hash_t un const char* :

```
#ifdef __PRESERVE_STRS__
    #define H_(X) (X)
    typedef const char* hash_t;
#else
    typedef unsigned long long hash_t;
    // compile-time hash
    extern constexpr hash_t H_(const char* str)
    {
        return details::hash_one(str[0], str + 1, details::basis);
    }
#endif
```

Du coup, partout où je sauvegarde des hash_t (formés par exemple depuis des paths avec H_("path/to/file.abc")), je peux si je le souhaite préserver ces strings en définissant **PRESERVE_STRS**, et je vois la string d'origine plutôt qu'un nombre énorme. Très utile pour le debug.

DEBUG & PROFILING OPTIONS

PROFILING_SET_2x2_TEXTURE -> Si défini, chaque Texture sera réduite à la taille 2x2. **PROFILING_SET_1x1_VIEWPORT** -> Si défini, une Texture render target sera réduite à la taille 1x1. **DEBUG_TEXTURE_VERBOSE** -> Si défini, certaines infos non critiques sont affichées dans le terminal.

[04-07-18]

[Bezier]

J'ai implémenté un algo de DeCasteljau pour le calcul stateless d'interpolations de Bézier, depuis un paramètre float et une liste de points (vec3). La liste peut être un parameter pack (perfect forwarding supporté) ou bien un std::vector.

[Texture]

J'ai complété la classe *Texture* qui peut maintenant être initialisée avec plusieurs textureID attachées à des samplers différents. Toute la partie image d'un *Material* peut être condensée dans un seul objet Texture. Le nom du fichier image sert à indiquer à quel sampler on doit adresser le textureID correspondant. Une image nommée cube_mt.diffuseTex.png sera associée au sampler2D mt.diffuseTex dans le shader ci-dessous:

```
struct material
{
    sampler2D diffuseTex;
    sampler2D specularTex;
    sampler2D normalTex;
};

uniform material mt;
```

La méthode Shader::update_uniform_samplers() effectue les calls suivants dans le bon ordre, pour chaque GLtexture de internal_ : glActiveTexture() glBindTexture() glGetUniformLocation() glUniform1i()

```
void Shader::update_uniform_samplers(const Texture& tex)
{
    for (uint8_t ii = 0; ii < tex.get_num_textures(); ++ii)
    {
        tex.bind(ii);
        GLint loc = glGetUniformLocation(ProgramID_,
           tex.get_sampler_name(ii).c_str());
        if (loc<0) continue;
        glUniform1i(loc, ii);
    }
}
```

On songera à stocker à l'avance les uniform locations…

[05-07-18]

[Texture]

La classe *Texture* peut maintenant être instanciée très facilement, au moyen d'un constructeur qui ne prend qu'un char* en argument (+ des params avec default). L'argument est un **nom d'asset**. Par exemple, dans le path suivant:

```
../res/textures/cube_mt.diffuseTex.png
```

"cube" est le nom de l'asset et "mt.diffuseTex" le nom du sampler associé. Donc si dans le dossier textures j'ai les fichiers:

```
cube_mt.diffuseTex.png
cube_mt.normalTex.png
cube_mt.specularTex.png
cube_mt.overlayTex.png
someShittyAsset_mt.diffuseTex.png
someShittyAsset_mt.normalTex.png
someShittyAsset_mt.specularTex.png
someShittyAsset_mt.overlayTex.png
...
```

J'ai toutes les infos avec seulement le nom de l'asset "cube" pour charger toutes les images qui vont bien, et en prime je peux sauvegarder les noms des samplers associés.

[Material]

J'ai un début de classe *Material* qui contient un Texture* et juste pour l'instant un float shininess. Material peut être initialisé avec un simple nom d'asset également grâce à la magie des constructeurs. La classe Shader peut maintenant envoyer en bloc tous les uniforms liés à un Material grâce à un call à update_uniform_material()

```
    Shader shader(/*...*/);
    Texture::load_asset_map(); // Read textures directory and store filenames
    Material cube_mat("cube"); // Load asset "cube" textures and props

    // ...

    // Inside game loop
    shader.use();
    shader.update_uniform_material(cube_mat);
```

On remarquera l'appel à Texture::load_asset_map() qui va parcourir le dossier textures et associer chaque nom d'asset aux paths des fichiers qu'il regroupe. J'utilise la libfs de c++17 (header).

Cmd arguments

Le programme test wcore peut être appelé avec l'argument "-s 800x600" pour ouvrir une fenêtre à cette résolution. Le format est "-s [scrWidth]x[scrHeight]".

[Texture] Named textures

La classe *Texture* possède une unordered_map statique dans laquelle on peut enregistrer des textures en leur donnant un nom, et récupérer une texture enregistrée en la cherchant par nom, via les méthodes:

```
static void register_named_texture(hash_t name, pTexture ptex);
static wpTexture get_named_texture(hash_t name);
```

avec

```
    typedef std::shared_ptr<Texture> pTexture;
    typedef std::weak_ptr<Texture> wpTexture;
```

La fonction register_named_texture() récupère l'ownership sur le shared_ptr qui lui est transmis (le shared_ptr sera scoped out côté appelant). En revanche la fonction get_named_texture() ne renvoie pas un pointeur shared mais un pointeur weak. Un weak_ptr doit être converti en shared_ptr avant utilisation:

```
    {
        auto ptex = std::make_shared<Texture>(screenWidth, screenHeight);
        Texture::register_named_texture(H_("screen"), ptex);
    }

    auto wptex = Texture::get_named_texture(H_("screen"));

    if (auto spt = wptex.lock()) { // Has to be copied into a shared_ptr before
        usage
        std::cout << spt->get_width() << "x" << spt->get_height() << std::endl;
    }
```

L'idée est que la classe Texture conserve l'ownership sur les textures nommées et n'a donc aucune raison de passer un shared_ptr.

UPDATE[H_]

Pas si géniale la macro qui ne fait rien. J'ai changé le feature debug **PRESERVE_STRS** comme suit:

```
#ifdef __PRESERVE_STRS__
    #define H_(X) (std::string(X))
    typedef std::string hash_t;
#else
```

afin de le préserver, j'ai eu des problèmes avec dans la classe de textures. Des unordered_map de char* … Qu'est-ce qui pouvait mal se passer ?

[Texture] Debbuging

La classe *Texture* ne configure plus des render targets pour CHAQUE putain de texture…

[TODO] Suite du programme

```
[x] Faire le rendu sur la texture nommée "screen"
[x] Ecrire de quoi l'afficher à l'écran (shader + support software pour dessiner
    un quad texturé)
```

[06-07-18]

Du lourd. On peut maintenant faire du rendu sur une *Texture* lors d'une première passe, et rendre un quad texturé avec celle-ci à l'écran lors d'une deuxième passe. La deuxième passe utilise un nouveau shader. Donc je peux faire du post-processing easy peasy et en théorie ça m'ouvre la voie à l'HDR rendering.

J'ai juste eu une galère en oubliant de déclarer et d'utiliser un VAO pour le quad… Noter que **l'utilisation de VAO n'est pas optionnelle** dans un contexte OpenGL moderne.

Pour l'instant tout traîne en vrac dans la classe main, il conviendra d'encapsuler tout ça dans du beau code OO.

Je ne sais pas encore si c'est bien ou pas bien, mais j'ai ajouté une méthode à *Texture* pour insérer automatiquement les bind / unbind du FBO sous-jacent autour de code de rendu, grâce à un foncteur :

```
    pscreen->with_render_target([&]()
    {
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // Bind VAO, draw, unbind VAO
        glBindVertexArray(VAO_);
```

```
        buffer_unit.draw(/* ... */);
        glBindVertexArray(0);
    });
```

Le même code hors du contexte d'exécution offert par le foncteur de with_render_target() dessinerait tout simplement à l'écran. J'ai trouvé ça esthétique… -> Edit: pas bien.

[BUG][fixed] Test overlay not displaying

L'overlay que j'utilise comme test pour les textures multiples n'est plus affiché quand j'utilise le rendu sur texture.

OUTPUT:

```
--------- FIRST PASS ----------
glActiveTexture GL_TEXTURE0+0    // shader.update_uniform_material
glBindTexture 0 -> id= 1
glUniform1i mt.diffuseTex 0
glActiveTexture GL_TEXTURE0+1
glBindTexture 1 -> id= 2
glUniform1i mt.overlayTex 1
--------- SECOND PASS ---------
glActiveTexture GL_TEXTURE0+0    // pscreen->bind
glBindTexture 0 -> id= 3
```

Fix temporaire -> Fix définitif

Commenter glBindTexture dans TextureInternal::bind_as_render_target().

```
    void Texture::TextureInternal::bind_as_render_target() const
    {
        //glBindTexture(GL_TEXTURE_2D, 0);
        glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer_);

        #ifdef __PROFILING_SET_1x1_VIEWPORT__
            glViewport(0, 0, 1, 1);
        #else
            glViewport(0, 0, width_, height_);
        #endif
    }
```

J'essaye de comprendre… Je crois que je n'ai simplement pas besoin d'appeler glBindTexture ici…

[PostProcessing]

J'ai implémenté les fonctions saturate() et gamma_correct() de mes vieux shaders sous WEngine dans le frag shader du quad, je peux faire du post processing basique (en l'occurrence, contrôler la "vibrance" et faire une correction gamma).

[TODO] Suite du programme

[x] Per fragment lighting [x] HDR lighting [x] Charger des .obj

“Physically based shading”

Commentaires intéressants sur: http://www.rorydriscoll.com/2013/11/22/physically-based-shading/

```
Sarcasmotron
 November 22, 2013 at 7:38 pm
Physically Based Shaders in the game context have the following scientific
   requirements:-

 energy conservation has to be mentioned somewhere, not necessarily enforced
    (because: art)-
 the specular falloff has to be messed around with somewhat, because Blinn-Phong
    is so passé-""
 gloss drives both specular exponent and reflection-
 use environment maps everywhere-
 it absolutely must have fresnel in it-
 gamma correction is performed-
 it must be praised as an amazing "next-"gen feature even though 'its technically
    something pretty trivial

The term "Physically "Based is required to make it abundantly clear that 'were
   performing some serious and important science here.
```

[07-07-18]

Aujourd'hui j'essaye d'attaquer le per fragment lighting. Idéalement je veux pouvoir gérer à terme 1 lumière directionnelle + plusieurs point-lights et éventuellement d'autres types de source comme les spot-lights. Donc je m'attends à avoir un fragment shader qui ressemble à ça:

```
    out vec4 FragColor;

    void main()
    {
      // define an output color value
      vec3 output = vec3(0.0);
      // add the directional light's contribution to the output
      output += someFunctionToCalculateDirectionalLight();
      // do the same for all point lights
      for(int i = 0; i < nr_of_point_lights; i++)
        output += someFunctionToCalculatePointLight();
      // and add others lights as well (like spotlights)
      output += someFunctionToCalculateSpotLight();

      FragColor = vec4(output, 1.0);
    }
```

source : https://learnopengl.com/Lighting/Multiple-lights

[ScreenRenderer]&[Scene]

Avant d'attaquer le lighting il m'a semblé plus sage d'organiser ce que j'avais déjà dans trois nouvelles classes.

ScreenRenderer condense tout le code lié de près ou de loin à la deuxième passe de rendu. Lors de sa construction il fabrique une *Texture* avec les render targets initialisées, enregistre cette texture comme **texture nommée “screen”**. Il initialise également un shader de rendu sur quad. Puis il construit un *Mesh* temporaire afin d'y stocker les

coordonnées normalisées des coins de l'écran et deux triangles. Il soumet ce mesh à son *BufferUnit* qui upload les vertices vers OpenGL. Enfin, il génère un VAO et initialise les attributs de vertex.

Sa fonction draw() ajuste le viewport, active le shader de rendu sur quad, update les deux uniforms de post-processing, bind la texture "screen" et update le sampler correspondant, puis bind son VAO et dessine, avant de restaurer l'état de GL.

Scene est pour l'instant une classe fourre-tout (comme souvent avec les classes de scène, à ce qu'il paraît), qui contient un vecteur de *Model* et une *Camera*. Elle possède un ensemble de fonctions traverse_models() et traverse_models_if() qui permettent à du code appelant d'itérer sur la collection de modèles en appliquant sur chacun d'entre eux un foncteur. Et dans le cas de traverse_models_if(), le foncteur est appliqué si et seulement si un prédicat en second argument évalue à true sur le modèle pointé (utile pour du *culling* j'ai pensé).

```
    scene.traverse_models([&](std::shared_ptr<Model> pmodel)
    {
        // Get model matrix and compute products
        mat4 M = pmodel->get_model_matrix();
        /* ... */
    });
```

Sa méthode update() prend un argument float dt, qui correspond au temps écoulé entre 2 frames (fourni par le code appelant quelque part dans le game loop). Cette méthode est responsable du mouvement des objets de la scène (changements de transformations etc…).

Sa méthode init()… initialise la scène (appelée dans le constructeur).

VertexArray est un bête wrapper autour des VAO d'OpenGL. Il doit être construit avec un *BufferUnit* en argument (VBO et IBO font partie de l'état du VAO). Sans surprise, on peut le bind() et l'unbind(). J'aurais très bien pu ne templater que le constructeur, mais je pense qu'en faisant ça j'aurais ouvert la porte à tout un tas de bugs de merde. Le système de typage m'empêchera de déconner trop fort. A l'avenir, j'ouvrirai la possibilité de le bind à plusieurs buffer units de même type.

[BUG][fixed] Pbm affichage terrain

Dans la scène j'ai mes 2 cubes animés. Quand je rajoute un modèle avec 4 vertices 3P3N2U (un quad plat texturé), le modèle apparaît collé aux cubes et les suit. J'ai remarqué ça en codant les terrains/heightmaps.

Les matrices sont bonnes, le buffer offset, les nombres d'indices et de vertices aussi. Tout se passe comme si le draw call des cubes affichait plus de vertices qu'il n'en faut.

```
// LOAD
// screen, osef
vert[(-1, -1, 0) (1, -1, 0) (-1, 1, 0) (1, 1, 0) ]
ind[0 1 2 1 3 2 ]

// cube 1
vertices.size()=24
indices.size()=36
transformed_indices.size()=36
vert[(0.5, 0, 0.5) (0.5, 1, 0.5) (-0.5, 1, 0.5) (-0.5, 0, 0.5) (0.5, 0, -0.5)
    (0.5, 1, -0.5) (0.5, 1, 0.5) (0.5, 0, 0.5) (-0.5, 0, -0.5) (-0.5, 1, -0.5)
    (0.5, 1, -0.5) (0.5, 0, -0.5) (-0.5, 0, 0.5) (-0.5, 1, 0.5) (-0.5, 1, -0.5)
    (-0.5, 0, -0.5) (0.5, 1, 0.5) (0.5, 1, -0.5) (-0.5, 1, -0.5) (-0.5, 1, 0.5)
    (0.5, 0, -0.5) (0.5, 0, 0.5) (-0.5, 0, 0.5) (-0.5, 0, -0.5) ]
ind[0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12 13 14 12 14 15 16 17 18 16 18 19 20
    21 22 20 22 23 ]
tr_ind[0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12 13 14 12 14 15 16 17 18 16 18 19
    20 21 22 20 22 23 ]
```

```
// cube 2
vertices.size()=24
indices.size()=36
transformed_indices.size()=36
vert[(0.5, 0, 0.5) (0.5, 1, 0.5) (-0.5, 1, 0.5) (-0.5, 0, 0.5) (0.5, 0, -0.5)
    (0.5, 1, -0.5) (0.5, 1, 0.5) (0.5, 0, 0.5) (-0.5, 0, -0.5) (-0.5, 1, -0.5)
    (0.5, 1, -0.5) (0.5, 0, -0.5) (-0.5, 0, 0.5) (-0.5, 1, 0.5) (-0.5, 1, -0.5)
    (-0.5, 0, -0.5) (0.5, 1, 0.5) (0.5, 1, -0.5) (-0.5, 1, -0.5) (-0.5, 1, 0.5)
    (0.5, 0, -0.5) (0.5, 0, 0.5) (-0.5, 0, 0.5) (-0.5, 0, -0.5) ]
ind[0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12 13 14 12 14 15 16 17 18 16 18 19 20
    21 22 20 22 23 ]
tr_ind[24 25 26 24 26 27 28 29 30 28 30 31 32 33 34 32 34 35 36 37 38 36 38 39 40
    41 42 40 42 43 44 45 46 44 46 47 ]

// "terrain"
vertices.size()=4
indices.size()=6
transformed_indices.size()=6
vert[(0, 0, 0) (1, 0, 0) (1, 0, 1) (0, 0, 1) ]
ind[0 3 2 0 2 1 ]
tr_ind[48 51 50 48 50 49 ]

// DRAW
Nind    Nvert   Boffset
36      24      0       // cube1
36      24      36      // cube2
6       4       72      // "terrain"
```

-> Tout va bien lors de l'upload.

J'ai le coupable (dans la boucle de rendu) :

```
    buffer_unit_.draw(pmodel->get_mesh().get_ni(),
                      pmodel->get_mesh().get_buffer_offset());
```

-> Remplacer par :

```
    buffer_unit_.draw(pmodel->get_mesh().get_ni()/3,
                      pmodel->get_mesh().get_buffer_offset());
```

Ne pas oublier de diviser par 3 le nombre d'indices pour avoir le nombre de triangles (d'éléments)

-> J'ai ajouté une méthode Mesh::get_n_elements() pour limiter la confusion à l'avenir.

Rq: Ca ne se voyait pas avec 2 cubes, parce que les vertices sont les mêmes. -> A l'avenir, faire des tests avec des géométries différentes le plus tôt possible.

[HeightMap] & mesh factory

J'ai récup ma bonne vieille classe *HeightMap* de WEngine, et l'ai intégrée au prix de quelques adaptations mineures. Une *HeightMap* possède une largeur (dimension x en model space) et une longueur (dimension z). A chaque x et z entier on associe une valeur float (on a un tableau de float inline doublement indexé):

```
    heights_[xx*length_+zz];
```

Comment alors trouver la hauteur pour des coordonnées non-entières ? Pour chaque valeur non entière on regarde dans quel quad on est, puis dans quel triangle de ce quad, puis à quelles coordonnées barycentriques dans ce triangle, puis on interpole la hauteur au moyen de ces dernières. Classique.

J'ai également deux visiteurs qui permettent d'adresser la height map comme un état d'automate cellulaire : traverse_4_neighbors() et traverse_8_neighbors() qui prennent des coordonnées entières et un foncteur en arguments. Le foncteur sera exécuté sur les 4 / 8 voisins du point en argument. Je me servais de ça pour les passes d'érosion de terrain dans WEngine.

Le namespace *factory* contient maintenant 2 méthodes d'initialisation de *Mesh* :

```
1     extern Mesh<Vertex3P3N2U>* make_cube_3P3N2U();
      extern Mesh<Vertex3P3N2U>* make_heightmap_3P3N2U(const HeightMap& hm);
```

Donc pour créer un cube texturé avec l'asset cube, on fait comme ça :

```
      pModel pcube = std::make_shared<Model>(factory::make_cube_3P3N2U(), "cube");
```

Pour l'instant mon terrain est un *Model* que j'envoie dans le même buffer unit que les cubes. Quand j'aurai besoin d'un traitement particulier pour les terrains, je ferai un renderer approprié.

Truc bizarre ?

Dans *Camera* j'ai du inverser la position donnée en argument à set_position, pour qu'un set_position avec un y positif ne m'envoie pas *sous* le sol.

```
1     inline void set_position(const math::vec3& newpos)
      { trans_.set_position(-newpos); }
```

[x] Réfléchir à ça à tête reposée.

[08-07-18]

12 Monkeys: https://fr.serie-streaming.cc/serie/12-monkeys/saison-3-episode-3-streaming.html

[Lighting]

J'ai implémenté du *per-fragment lighting* avec un modèle de *Phong* basique. Tout fonctionne comme prévu.

Au passage, j'ai corrigé un bug mineur. Dans Shader::update_uniform_material() c'est là que je bind les textures. Je viens de rajouter en première instruction dans cette méthode :

```
      glBindTexture(GL_TEXTURE_2D, 0);
```

L'idée est que si je n'unbind jamais les textures et que j'ai deux modèles avec des assets ne comprenant pas le même nombre de sous-textures, les sous-textures sur-numéraires resteront bound d'un draw call à l'autre. Donc si j'ai d'une part un sol avec une texture diffuse et une spéculaire, et d'autre part un cube avec une diffuse, une spéculaire plus un overlay, l'overlay sera aussi appliqué au sol pendant le rendu, puisqu'il est resté bound…

[Texture][OPT]

Si le texture binding/unbinding coute trop cher, sauvegarder dans le shader le nom d'asset du dernier material qui est bound, puis si un call à update_uniform_material() veut pousser un material différent, là seulement on unbind.

[Light] & [DirectionalLight]

La classe *Light* est une classe de base pour tous les types de sources lumineuses. Son seul membre est la position de la source. Elle possède une méthode virtuelle pure :

```
    virtual void update_uniforms(unsigned int program_id) const = 0;
```

La classe *DirectionalLight* hérite de *Light* et implémente un override pour update_uniforms(), où les uniforms correspondant à la lumière directionnelle sont updatés.

Shader possède maintenant une méthode pour updater les uniforms de n'importe quel type de lumière en exploitant le polymorphisme dynamique :

```
    void Shader::update_uniform_light(const Light& light,
                                      const math::vec3& eye)
    {
        light.update_uniforms(ProgramID_);
        GLint loc_vp = glGetUniformLocation(ProgramID_, "rd.v3_viewPos");
        glUniform3fv(loc_vp, 1, (GLfloat const*)&eye);
    }
```

Je peux maintenant balancer tous mes types de lumière dans un container dans la scène, et implémenter un visitor pour traverser toutes les lumières. Du coup, je peux facilement automatiser l'envoi d'uniforms pour les lumières de la scène. J'évite en général d'avoir recours au polymorphisme dans ce projet à cause de l'overhead, mais je m'attends à ce qu'il n'y ait qu'un nombre restreint de lumières dans la scène. Boom. Chose faite.

[Game loop]

J'ai réhabilité la classe *Clock* de WEngine et implémenté un game loop plus sophistiqué dans *GLContext*. Cette dernière classe contient maintenant 3 foncteurs initialisables (setup, update et render). Toutes les updates du jeu se font à travers :

```
    context._update([&](float dt)
    {
        scene.update(dt);
        /* ... */
    });
```

La méthode render remplace l'ancienne méthode loop. Le game loop utilise des chronomètres nanosec pour mesurer le temps écoulé dans la boucle et décider du temps à attendre afin d'atteindre un FPS cible. Si de plus l'option de compilation **PROFILING_GAMELOOP** est définie, alors un chronomètre supplémentaire va mesurer les durées prises par chaque sous-ensemble de la boucle (pour l'instant seulement le rendering). Un rapport est présenté dans la console, et je me sers de séquences ANSI pour contrôler la position du curseur et éviter que la console ne défile comme une guedin :

```
       -->    renderedTexture ID=2 Loc=2
    [*]   [Texture] Registering new named texture: screen
  [TRACK] -------- Game loop start --------
              Frame:    20082.119852µs
              Active:   268.981996µs    1.339410%
              Idle:     19731.018692µs  98.251671%
              Render:   120.053999µs    0.597815%
  [TRACK] -------- Game loop stop --------
    [*]   Destroying shader program [7].
```

On a les durées pour la frame entière (Frame), le temps actif CPU (Active), le temps inactif (Idle) et le temps de rendu (Render). Les pourcentages sont calculés par rapport à la Frame.

[ANSI Escape]

http://wiki.bash-hackers.org/scripting/terminalcodes http://ascii-table.com/ansi-escape-sequences.php

```
**\033[5A** -> Remonte le curseur de 5 lignes
**\033[2K** -> Efface la ligne courante en entier.
**\033[1;38;2;255;100;0m** -> Ecrit en orange (255,100,0).
**\033[0m** -> Restaure le style par défaut.
```

[BUG][fixed] Crash graphique lourdingue

J'essaye d'implémenter les point lights, mais dès que je modifie le shader j'ai des bugs graphiques à la con, genre terrain qui disparaît, cube qui disparaît, cube avec un terrain attaché… Parfois c'est un ralentissement complet, parfois c'est un freeze du GUI de Linux. C'est pour l'instant incompréhensible. Par modifier, j'entends à peu près tout et n'importe quoi : - Inverser deux membres d'une struct - Ajouter une bête fonction - Faire un calcul supplémentaire - Ajouter / retirer du code **commenté** Tant que les uniforms de la point light sont poussés, j'ai la possibilité d'un bug.

Voir si c'est relié : le putain de bug avec hud_service de merde sous Ubuntu 16.04.

De toute façon j'envisage de tout bazarder pour commencer une pipeline PBR, il est temps de faire de nouvelles choses.

UPDATE

Non, y a de toute évidence un problème avec la classe de textures… Quand je change l'asset d'un des deux cubes pour brickWall, le sol ne s'affiche plus. Quand je rajoute un fichier de texture spéculaire pour l'asset brickWall, le problème est réglé. Hmmm… C'est overlay qui fout le bordel ! On a un **uniform non initialisé** !

Idées pour les uniforms de PointLights multiples

https://learnopengl.com/Lighting/Multiple-lights dans les coms :

```
    for (GLuint i = 0; i < 4; i++)
    {
    string number = to_string(i);

    glUniform3f(glGetUniformLocation(ObjectShader.Program, ("pointLight[" + number
        + "].position").c_str()), PointLightPosition[i].x, PointLightPosition[i].y,
        PointLightPosition[i].z);
    glUniform3f(glGetUniformLocation(ObjectShader.Program, ("pointLight[" + number
        + "].ambient").c_str()), pointLightColors[i].r * 0.1f,
        pointLightColors[i].g * 0.1f, pointLightColors[i].b * 0.1f);
    glUniform3f(glGetUniformLocation(ObjectShader.Program, ("pointLight[" + number
        + "].diffuse").c_str()), pointLightColors[i].r, pointLightColors[i].g,
        pointLightColors[i].b);
    glUniform3f(glGetUniformLocation(ObjectShader.Program, ("pointLight[" + number
        + "].specular").c_str()), 1.0f, 1.0f, 1.0f);
    glUniform1f(glGetUniformLocation(ObjectShader.Program, ("pointLight[" + number
        + "].constant").c_str()), 1.0f);
    glUniform1f(glGetUniformLocation(ObjectShader.Program, ("pointLight[" + number
        + "].linear").c_str()), 0.09f);
    glUniform1f(glGetUniformLocation(ObjectShader.Program, ("pointLight[" + number
        + "].quadratic").c_str()), 0.032f);
    }
```

Aussi, pour changer dynamiquement la taille du tableau de point lights d'un frag shader comme ceci :

```
    #define NR_POINT_LIGHTS 4
2   uniform PointLight pointLights[NR_POINT_LIGHTS];
    /* ... */
```

On peut générer dynamiquement une string

```
1   std::string defines("#define NR_POINT_LIGHTS ");
    defines += std::to_string(n_lights);
```

puis concaténer cette string à la string source du fragment shader avant compilation. A chaque changement de scène, on peut recalculer le nombre de lumières dont on a besoin et compiler un shader à la volée.

[TODO] Suite du programme : Going full PBR

J'ai pu aujourd'hui valider un certain nombre de features en codant vite taif un Phong model. Maintenant je dois attaquer le plat de résistance.

```
[o] Ecrire du code en C++ qui implémente les fonctions dont on a besoin dans les
    shaders PBR et tester à blinde. Je pourrai me servir de mes classes de maths
    pour reproduire les types GLSL.
2 [x] Ecrire les shaders en se basant sur le code précédent.
```

[09-07-18]

Déjà on commence par régler le problème rencontré hier. Je vais pousser les instrus de debug pour les uniforms et automatiser la collection de leur noms.

[DEBUG] Notes de session

- Ne pas envoyer les uniform samplers ids en unsigned int avec un type sous-jacent uint_8, ça fout évidemment le bordel.
- Définir **DEBUG_SHADER** pour surveiller de près ce qui se passe avec les uniforms.
- Méthodes Light::update_uniforms() modifiées pour prendre un const Shader& en argument. Les noms d'uniforms échappent au seul scope de la classe Shader mais c'est un moindre mal. Tous les calls sont centralisés dans la classe *Shader* et c'est moins lourdingue à débugger.
- Je crois que je suis vraiment con. Faut active/bind les textures juste avant le draw call. Je ne le fais pas, donc ce qui s'affiche provient au mieux d'un état rémanent non défini. -> J'ai fait une méthode Texture::bind_all() à appeler avant le draw call, dans la boucle de traverse des Models. -> Si on s'assure que tous les assets définissent les textures dont a besoin un shader donné, on n'a plus de problème.
- Y a pas qu'un problème de texture (qui semble réglé maintenant). Quand je rajoute un 3ème cube dans la scène, je ne vois plus le sol. Quand j'inverse l'ordre de déclaration entre le sol et le cube 3, je vois le sol à nouveau. J'ai l'impression que le moteur vit assez mal d'avoir des géométries différentes dans le même buffer unit, même si ça devrait pas poser problème.
  - On a un freeze en général quand on dépasse une taille de terrain de 32x32.
  - Si cubes avant terrain -> terrain disparaît et inversement.
  - La position du terrain semble jouer (vérifier que ce n'est pas simplement parce qu'il clip comme un gros connard).

On le voit :

```
    terrain_patch_->set_position(vec3(-4.0,-2.0,-4.0));
```

On le voit plus :

```
1     terrain_patch_->set_position(vec3(-4.0,-2.0,-5.0));
```

- Dans *Texture*, j'initialise par défaut les filtres avec GL_LINEAR_MIPMAP_LINEAR. Ca a pour effet d'initialiser GL_MAG_FILTER (et GL_MIN_FILTER) à GL_LINEAR_MIPMAP_LINEAR, ce qui renvoie une erreur GL_INVALID_ENUM. Si je configure par défaut en GL_LINEAR / GL_NEAREST… quoi que ce soit d'autre de valide pour GL_MAG_FILTER, alors le temps de swap est **mega** ralenti, même pour ma scène ridicule.

glGetError

```
1 1280 GL_INVALID_ENUM
  1281 GL_INVALID_VALUE
3 1282 GL_INVALID_OPERATION
  1283 GL_STACK_OVERFLOW
5 1284 GL_STACK_UNDERFLOW
  1285 GL_OUT_OF_MEMORY
```

[BUG][fixed] Instabilité shader

Quand je passe les shaders de 400 core à 410, un cube disparaît…

[10-07-18]

Il semblerait que j'ai réglé simultanément : * le problème du crash graphique * de la disparition de géométrie * du ralentissement lors du passage des textures en GL_LINEAR * l'instabilité que j'attribuais aux shaders.

Le membre Mesh::buffer_offset_ **DOIT** être initialisé à 0 dans le constructeur, malgré l'appel à set_buffer_offset() au moment du BufferUnit::submit(). En théorie on set avant de get, mais ça fout quand même le bordel pour une raison qui m'échappe. Le fait est qu'initialiser proprement le constructeur fait le boulot. J'ai bien fait de persister, ce bug de merde m'aurait suivi quoi que je fasse.

TOUJOURS INITIALISER LES SCALAIRES A LA CONSTRUCTION DES TYPES COMPOSITES

[Lighting]

Lumière ponctuelle opérationnelle. J'ai une scène avec 3 cubes animés, un terrain, une lumière directionnelle et 3 lumières ponctuelles qui suivent des courbes de Bézier.

[DEBUG] Memory leaks

```
>> valgrind --leak-check=full -v ../bin/wcore
```

```
1 ==17785== HEAP SUMMARY:
  ==17785==     in use at exit: 477,800 bytes in 2,973 blocks
3 ==17785==   total heap usage: 26,831 allocs, 23,858 frees, 223,545,489 bytes
     allocated
  ==17785==
5 ==17785== Searching for pointers to 2,973 not-freed blocks
  ==17785== Checked 4,775,792 bytes
7 ==17785==
  ==17785== 72 bytes in 1 blocks are definitely lost in loss record 84 of 118
```

```
==17785==    at 0x4C2FB55: calloc (in
   /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==17785==    by 0x85498A0: XkbGetMap (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==17785==    by 0x54892FA: ??? (in /usr/lib/x86_64-linux-gnu/libglfw.so.3.1)
==17785==    by 0x54859BC: glfwInit (in /usr/lib/x86_64-linux-gnu/libglfw.so.3.1)
==17785==    by 0x48D7EF: GLContext::GLContext(unsigned int, unsigned int) (in
   /home/ndx/Desktop/WCore/bin/wcore)
==17785==    by 0x460E19: main (in /home/ndx/Desktop/WCore/bin/wcore)
==17785==
==17785== LEAK SUMMARY:
==17785==    definitely lost: 72 bytes in 1 blocks
==17785==    indirectly lost: 0 bytes in 0 blocks
==17785==      possibly lost: 0 bytes in 0 blocks
==17785==    still reachable: 477,728 bytes in 2,972 blocks
==17785==         suppressed: 0 bytes in 0 blocks
==17785== Reachable blocks (those to which a pointer was found) are not shown.
==17785== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==17785==
==17785== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==17785== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

glfw/x11 memory leak, génial.

[11-07-18]

Je ne suis pas content de ma classe *Texture* qui est devenue un énorme blob. Je vais refactor cette merde. Je pense qu'il faut séparer les fonctionnalités de rendu sur texture (frame buffer & render buffer). On aurait un objet *FrameBuffer* contenant un render buffer non initialisé, et un objet *Texture*. Le frame buffer pourrait être attaché à une texture, en précisant les attachments (un par texture unit), et si aucun GL_DEPTH_ATTACHMENT n'est spécifié pour une texture unit, alors un render buffer serait initialisé pour servir de depth buffer, afin que le frame buffer soit complet.

[FrameBuffer]

La classe *FrameBuffer* reprend tout le code de *Texture* et *TextureInternal* qui concerne les render targets. Elle contient un index de frame buffer et un index de render buffer. A la construction, on lui passe une *Texture* à attacher, ainsi qu'un std::vector d'attachments (peut être initialisé avec un {„} dans un constructeur).

La classe *ScreenRenderer* a été modifiée en conséquence et donne accès au contexte de son membre *FrameBuffer* via une méthode with_frame_buffer_as_render_target() qui prend un foncteur en argument et forwardera celui-ci à FrameBuffer::with_render_target().

Voilà qui préparera la voie au *deferred shading*. L'idée derrière le deferred shading est de réaliser une première passe de rendu dans une texture (plusieurs units) appelée le *G-buffer* pour y écrire l'information géométrique brute (*geometry pass*). Une deuxième passe (*lighting pass*) utilise l'information du G-buffer pour éclairer chaque fragment. Avantages princpiaux : * La deuxième passe travaille sur un quad, donc autant de fragments que de pixels. En effet, la passe précédente a déjà depth-testé et l'information géométrique qui subsiste est l'information *visible* (top-most fragments). De fait le lighting est beaucoup plus économique, car le fragment shader ne travaillera pas sur des fragments invisibles comme c'est le cas en forward rendering. * L'utilisation de *light volumes* permet de traiter une large quantité de lumières sans effondrement des performances. L'idée est qu'on effectue les calculs de lumières pour un fragment donné et une source donnée ssi le fragment est dans le light volume, c'est à dire la zone d'influence, de la source.

Inconvénients: * Incompatible avec le MSAA. * Rend impossible le blending (qui devra toujours être réalisé en forward).

[Texture] & [FrameBuffer]

Il est maintenant possible d'attacher plusieurs color buffers à un FBO. La *Texture* est initialisée avec comme premier argument un vecteur de strings désignant les noms des samplers correspondant à chaque texture unit dans l'ordre. Le *FrameBuffer* est initialisé avec en argument la texture nouvellement créée, et un vecteur de GLenum désignant les attachments :

```
ptex = std::make_shared<Texture>(
    std::vector<std::string>{"positionTex", "normalTex"},
    screenWidth,
    screenHeight,
    GL_TEXTURE_2D,
    GL_NEAREST,
    GL_RGB,
    GL_RGB,
    false);

FrameBuffer FBO(*ptex, {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1}),
```

- La texture unit 0 est référencée par le sampler2D positionTex dans les shaders qui l'échantillonnent (texture en input), et correspond à l'attachment couleur 0 (donc à la variable layout(location = 0) out vec3 out_position) dans les shaders qui écrivent dessus (texture en output).
- La texture unit 1 est référencée par le sampler2D normalTex dans les shaders qui l'échantillonnent (texture en input), et correspond à l'attachment couleur 1 (donc à la variable layout(location = 1) out vec3 out_normal) dans les shaders qui écrivent dessus (texture en output).

C'est ainsi que je compte faire le deferred shading : on génère une *Texture* qui servira de g-buffer, avec autant de samplers que nécessaire (position, normal, albedo, specular) et un *FrameBuffer* initialisé avec cette texture et autant d'attachments. Une seule passe de rendu écrira simultanément dans tous les color buffers et le render buffer (depth-buffer). La passe suivante n'aura qu'à sampler cette même texture pour retrouver l'information géométrique de la scène.

[12-07-18]

J'implémente tout ce qu'il faut pour le rendu différé.

[GBuffer]

La classe *GBuffer* possède un pointeur sur *Texture* nommée (gbuffer), un *FrameBuffer* initialisé avec cette texture, et un *Shader* pour la passe géométrique. La texture gbuffer se fait fourrer des données par le frame buffer au moyen de 3 attachments : * GL_COLOR_ATTACHMENT0 <- position (world space), sampler = "positionTex" * GL_COLOR_ATTACHMENT1 <- normal (world space), sampler = "normalTex" * GL_COLOR_ATTACHMENT2 <- albedo ET specular (world space), sampler = "albedoSpecTex"

Position et normal sont codés en **half-float** (16 bits). Albedo (diffuse color) et intensité spéculaire (scalaire) partagent un seul color attachment en RGBA. La partie RGB contient la partie diffuse, et A l'intensité spéculaire (canal rouge de la texture spéculaire).

J'ai modifié la classe *ScreenRenderer* pour qu'elle affiche ces trois textures après une passe géométrique sur la scène (ssi le **DEBUG_GBUFFER** est défini) dans 3 petits quads en haut de l'écran.

Lors de l'écriture du shader de la passe géométrique, j'ai observé qu'OpenGL ne supportait pas que je définisse des structs en entrée / sortie et optimisait normale et position quoi que je fasse, donc j'ai dû sortir les données vecteur par vecteur :

```
    // gpass.vert
    out vec3 vertex_pos;
    out vec3 vertex_normal;
    out vec2 vertex_texCoord;
    // ...
```

Je parviens donc à écrire dans les attachments sans problème. Maintenant je dois écrire le shader de la passe d'illumination et sampler le g-buffer pour obtenir les positions / normales / uv dans le repère monde.

[Defered Shading]

Great Success!

Donc pour résumer ce que j'ai à l'instant, on utilise le *Shader* "gpass" afin de rendre la scène sur la texture nommée "gbuffer" via l'objet *GBuffer*. Ce shader écrit les données géométriques de la scène dans 3 color attachments (et des infos pour le debug dans un 4ème). Ensuite, c'est un quad (2 triangles) qui est rendu sur la texture nommée "screen" au moyen du shader "lpass" qui réalise la passe d'illumination. lpass définit les samplers déclarés dans g_buffer.cpp ("positionTex", "normalTex", "albedoSpecTex", "debugTex") et échantillonne ces textures pour obtenir les vecteurs pour le calcul de lumière (et l'overlay debug):

```
    uniform sampler2D positionTex;
    uniform sampler2D normalTex;
    uniform sampler2D albedoSpecTex;
    uniform sampler2D debugTex;

    void main()
    {
        vec3 fragPos = texture(positionTex, texCoord).rgb;
        vec3 fragNormal = texture(normalTex, texCoord).rgb;
        vec3 fragAlbedo = texture(albedoSpecTex, texCoord).rgb;
        float fragSpecular = texture(albedoSpecTex, texCoord).a;
        float fragOverlay = texture(debugTex, texCoord).r;
        float fragWireframe = texture(debugTex, texCoord).g;

        vec3 viewDir = normalize(rd.v3_viewPos - fragPos);
        vec3 directional_contrib = calc_dirlight(dl, fragNormal, viewDir,
           fragAlbedo, fragSpecular);
        // ...
    }
```

J'ai modifié update_uniform_material() pour ne plus envoyer l'uniform rd.f_shininess dont je ne me sers plus (pas la peine, je passe en PBR bientôt). J'ai hardcodé cette valeur dans lpass.frag.

La classe *ScreenRenderer* est devenue *ScreenBuffer* par souci d'homogénéité avec *GBuffer* qui possède une mécanique similaire. Une interface commune est envisagée.

[HDR]

Great success!

La *Texture* nommée "screen" devient un *floating point buffer* (internal format = GL_RGBA16F) ce qui permet aux valeurs RGB de dépasser l'intervalle [0,1]. Le quad shader responsable du post-processing réalise une étape de *tone mapping* juste avant. L'algo de tone mapping utilisé permet d'introduire un paramètre d'*exposition* contrôlable dynamiquement :

```
void main()
{
```

```
    // "screen" texture is a floating point color buffer
    vec3 hdrColor = texture(diffuseTex, texCoord).rgb;
    // Exposure tone mapping
    vec3 mapped = vec3(1.0) - exp(-hdrColor * rd.f_exposure);
    // Color saturation
    mapped = saturate(mapped, rd.f_saturation);
    // Gamma correction
    out_color = gamma_correct(mapped, rd.v3_gamma);
}
```

[sRGB]

Les *Textures* **diffuses** sont maintenant chargées avec un internal format GL_SRGB_ALPHA. L'idée est que les textures diffuses ont été produites en se servant d'un moniteur, et sont donc déjà dans l'espace sRGB (comme si elles avaient subit une correction gamma). Donc si on les load en RGBA, le post processing va appliquer une correction gamma "supplémentaire" et les textures diffuses paraitront anormalement claires. OpenGL propose un format interne sRGB qui permet d'éviter cet écueil. En revanche, il faut bien veiller à ce que les autres textures (normal, specu…) soient toujours loadées en RGB/RGBA, parce qu'elles sont définies dans l'espace linéaire RGB.

[Lighting][reference] Point lights coefficients

Table des coeffs pour les scalaires K0, K1 et K2 d'une *PointLight* en fonction de la *Distance* illuminée maximale souhaitée :

```
*Distance*  *Constant*  *Linear*  *Quadratic*
    7          1.0       0.7        1.8
    13         1.0       0.35       0.44
    20         1.0       0.22       0.20
    32         1.0       0.14       0.07
    50         1.0       0.09       0.032
    65         1.0       0.07       0.017
    100        1.0       0.045      0.0075
    160        1.0       0.027      0.0028
    200        1.0       0.022      0.0019
    325        1.0       0.014      0.0007
    600        1.0       0.007      0.0002
    3250       1.0       0.0014     0.000007
```

Sources :

```
[1] https://learnopengl.com/Lighting/Light-casters
[2] http://wiki.ogre3d.org/tiki-index.php?page=-Point+Light+Attenuation
```

[TODO] Suite du programme

```
[x] Bloom effect
[x] Light volumes
    [x] Using if statement in lpass.frag
    [x] Using sphere meshes
```

[13-07-18]

Aujourd'hui j'essaye d'implémenter Bloom.

J'ai vu une technique naïve qui consiste à générer une texture *bright pass* (seuillage en fonction de l'intensité lumineuse) puis à convoluer itérativement cette texture avec un noyaux Gaussien 5x5 en alternant passe verticale et passe horizontale (principe du filtre séparable). On obtient la version floutée de la texture bright pass, qu'on peut combiner avec le rendu de la scène en post-processing, par *additive blending*. Ceci peut être implémenté à l'aide de deux frame buffers qui se renvoient la texture après chaque passe. L'un convolue verticalement et l'autre horizontalement. Un cas typique de *ping-pong buffers*.

Un inconvénient majeur de cette approche est qu'il faut itérer longtemps avant d'obtenir suffisamment de blur, et le flou obtenu est très diffus et manque d'intensité. De plus on itère sur tous les pixels d'une texture de la même résolution que l'écran, ce qui est couteux.

source : https://learnopengl.com/Advanced-Lighting/Bloom

Une autre approche consiste à *approximer* des noyaux plus gros que 5x5 en exploitant le *downscaling* et le *filtrage bilinéaire* sur la texture bright pass.

En effet, mettons que l'on commence avec une texture 128x128. On downscale cette texture en 64x64, puis on applique un noyau 5x5 avant de rescale en 128x128 avec filtrage bilinéaire. Le résultat est comparable à l'action d'un noyau 11x11 sur la texture 128x128 d'origine. Si l'on répète l'opération sur des versions downscalées à 32x32 et 16x16 on obtient respectivement les approximations de noyaux 21x21 et 41x41. En rescalant toutes les textures à la taille d'origine et en les combinant par additive blending (éventuellement avec pondération), on obtient un flou beaucoup plus intéressant artistiquement, et les opérations qui simulent des noyaux plus en plus gros sont en réalité de plus en plus rapides (en 1/N^2) car appliquées à des textures plus petites.

On pourrait donc initialiser une texture bright pass avec une résolution valant la moitié de celle de l'écran (ou mieux (?) des tailles puissances de 2 grâce à math::np2() ou math::pp2()), et la downscale N-1 fois dans N-1 textures. Sur les N textures obtenues on calcule les convolutions, puis on rescale tout à la résolution de l'écran avant de blend.

L'utilisation de mipmaps pourrait (?) automatiser l'étape de downscaling.

Sources :


```
[1] http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/
[2] https://software.intel.com/en-us/articles/compute-shader-hdr-and-bloom
[3] http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with
-linear-sampling/
```


[Zik]

Occams Laser - Illumination

[Bloom]

–LIGHTING PASS– … –BRIGHT PASS– * La bright pass prend en entrée la texture écrite par la lighting pass, et écrit sur une texture initialisée avec 4 niveaux de mipmaps. –BLUR PASS– * FBO0 est initialisé à la taille originale, et réalise une blur pass avec le niveau 0 dans une texture T0 de la même taille. * FBO1 est initialisé à la taille 1/2, et réalise une blur pass avec le niveau 1 dans une texture T1 de taille 1/2. … * FBO3 … –POST PROC– T0, T1, T2 et T3 sont passées au shader de post-processing qui réalise le blending.

[ProcessingStage]

Comme mentionné hier dans [Defered Shading], il m'a semblé essentiel de regrouper les fonctionnalités de *ScreenBuffer* et *GBuffer* derrière une classe de base *ProcessingStage*. La fonction draw() est virtuelle avec une implémentation de base.

[Bright Pass] sorta...

La classe *ScreenBuffer* initialise maintenant une deuxième texture unit (sampler "brightTex") en GL_LINEAR_MIPMAP_LINE avec l'option (nouvelle) lazy_mipmap = true dans le constructeur de *Texture* (ce qui veut dire que les mipmaps ne sont pas générées lors de la construction de la *Texture*). Dans ScreenBuffer::draw() juste après avoir rendu le quad texturé ce qui a produit la bright map, les mipmaps de la bright map sont générés.

Donc plutôt que d'implémenter une bright pass en rendant le quad à nouveau, je me sers du *multiple render target trick* pour générer la bright map dès la lighting pass, et le rescaling de la bright map en textures de tailles sous-multiples pour l'étage blur est géré automatiquement par le mécanisme des mipmaps. Nice!

[BlurPass]

La classe *BlurPass* comporte un vecteur de textures (tailles 1, 1/2, 1/4, 1/8) nommées "bloom_ii" avec ii in [0,3], et un vecteur de *FrameBuffer*. Le frame buffer d'indice ii est initialisé avec la texture "bloom_ii" correspondante. BlurPass::generate_bloom_textures() est supposée générer les textures "bloom_ii" au moyen de la bright map.

Cependant, à chaque appel à fbo_[ii].with_render_target(&) on a une erreur GL_INVALID_OPERATION et *ScreenBuffer* qui est supposé pouvoir afficher les 4 textures bloom en mode debug (ssi **DEBUG_BLOOM** est défini) affiche de la merde. FUUUUUUUCK. 4h que je suis sur cette connerie.

[14-07-18]

Rien de bien difficile à corriger, je me plantais simplement dans le binding des textures à cause de la fatigue.

Bloom fonctionne dans les grandes lignes mais doit être amélioré. Je ne fais qu'une passe horizontale sur 4 textures "bloom_ii" avec 4 FBO différents. Je crois me souvenir qu'il est plus rapide de switcher les ressources attachées au FBO que le FBO lui-même (ce que ne permet pas encore ma classe FBO), donc c'est une approche couteuse, d'autant plus qu'il me faudrait quoi, 4 FBOs supplémentaires pour la passe verticale ?

[Bloom]

Donc c'est ce que j'ai pour l'instant : 8 FBOs pour l'effet bloom. Je ne peux pas utiliser le multiple render target trick pour générer les 4 channels de bloom en même temps, car un frame buffer attaché à des textures de tailles différentes limite la zone de rendu à la taille de la plus petite :

```
If the attachment sizes are not all identical, rendering will be limited to the
largest area that can fit in all of the attachments (an intersection of rectangles
having a lower left of (0; 0) and an upper right of (width; height) for each
attachment).
```

[BUG][fixed]

Ensuite, j'ai *parfois* des artéfacts sur les bords de l'écran après les passes blur. J'ai observé que tous les canaux n'en ont pas. -> Clamper les textures bloom semble largement amoindrir le problème.

Textures bloom de taille puissance de 2

J'ai laissé l'option d'utiliser des textures bloom de tailles puissance de 2 (ssi **OPTIM_BLOOM_USE_PP2** est défini). *Activer cette option aura pour conséquence d'étirer verticalement le halo.* Pourquoi ai-je seulement fait ça ?!

[Command Line Galore]

Obtenir la résolution de l'écran :

```
>> xdpyinfo | grep dimensions
```

[TODO] Suite du programme

Court terme :

```
[x] Implémenter des contrôles basiques pour bouger dans la scène.
[x] Implémenter du rendu de texte (FreeType a priori).
[x] Implémenter du parsing XML basique pour pouvoir construire des scènes test
   plus rapidement. Ce sera provisoire, donc ne pas y passer trop de temps.
    ->[23-07-18] Je risque de tout faire avec Flatbuffers.
```

Moyen Terme :

```
[ ] Implémenter les fonctionnalités suivantes et les shortcuts qui vont avec :
    [x] Figer / Reprendre les mouvements dans la scène
    [x] Activer / Désactiver certains systèmes
    [x] Afficher / Masquer les bounding boxes des objets
    [x] Afficher / Masquer le wireframe
    [ ] Ajouter / Supprimer / Cloner des objets de la scène
[x] Implémenter du ray casting pour pouvoir sélectionner des objets in-game.
    [ ] Utiliser un stencil test pour mettre en surbrillance le contour d'un objet
       sélectionné.
[ ] Converger vers un éditeur de niveaux in-game basique avec :
    [ ] Undo / Redo
    [ ] Copy / Paste
    [ ] Load / Save
[ ] Si l'on a besoin d'un GUI avec plein de contrôles différents :
* Pourquoi pas en faire une application séparée qui stream les données vers le
   moteur (*inter-process communication*) ?
    [ ] Au cas où je me pose la question, ce sera le GUI qui possèdera
       l'*authoritative state*. L'éditeur stream les paramètres quand on les
       modifie, le jeu update ses paramètres propres pour s'aligner avec l'état de
       l'éditeur. Et c'est un thread séparé qui communique avec le GUI par IPC.
* Ou bien inclure un render target dans une app en Python et utiliser SWIG (réputé
   problématique) / SIP pour générer les wrappers autour d'une API C-like du
   moteur (ou boost::Python et une API C++, pourquoi pas, c'est étonnamment
   user-friendly pour une lib boost). Cppyy a aussi l'air intéressant.
```

Long terme :

```
[x] Choisir et implémenter un algo de partition de l'espace (BSP, octree, k-D
   tree).
[ ] Ecrire un renderer pour cette structure de données.
[ ] Ecrire des classes pour (dé)sérialiser cette structure (level loading).
```

Sources :

```
thème général :
[1] https://gamedevelopment.tutsplus.com/articles/
make-your-life-easier-build-a-level-editor--gamedev-356
BSP chez Valve :
```

```
[2] https://developer.valvesoftware.com/wiki/Source_BSP_File_Format
[3] https://developer.valvesoftware.com/wiki/Brush
[4] http://www.flipcode.com/archives/Quake_2_BSP_File_Format.shtml
C++ Python wrapping:
[5] http://intermediate-and-advanced-software-carpentry.readthedocs.io
/en/latest/c++-wrapping.html
[6] https://www.boost.org/doc/libs/1_49_0/libs/python/doc/tutorial/doc
/html/python/exposing.html
[7] http://cppyy.readthedocs.io/en/latest/
```

[16-07-18]

J'ai codé des contrôles pour bouger la caméra (forward/backward, strafe left/right, move up/down, rotate yaw/pitch). J'ai dû virer la *Transformation* de *Camera*, et stocker séparément le yaw et le pitch dans deux float, afin de ne pas générer de roll (et ça simplifie la contrainte sur le pitch). Le quaternion en bougeant se dénormalise à peine, et l'erreur engendrée suffit à générer du roll, c'est pareil quand on stock une seule matrice. Pour une FPS cam, on n'a pas vraiment le choix que de séparer ces composantes, quite à produire un quat quand on en a besoin. Ici je génère directement une matrice pour aller plus vite. Donc *Transformation* ne me sert plus à rien dans *Camera*. De plus pour ajuster le yaw, la caméra doit tourner autour de l'axe (0,1,0) qui ne lui est pas propre. En effet, si le repère caméra est orthogonal (left,up,lookat) ou (right,up,-lookat), alors le vecteur up n'est pas le vecteur (0,1,0). Par ailleurs, le repère (left,up,lookat) qui semble si naturel est en flip-rotation par rapport à celui d'OpenGL. Il s'ensuit que pour construire la matrice view, j'ai du bricoler comme un connard, faute de vouloir y réfléchir pour de vrai :

```
    mat4 R;
    init_rotation_euler(R, 0.0f, -TORADIANS(yaw_), -TORADIANS(pitch_)); // WTF
        minus?!
    mat4 T;
    init_translation(T, position_);
    return R*T; // T then R (because R transposed ?!)
```

Et pour récupérer les vecteurs left et forward, on doit regarder les *lignes* de la matrice view et pas les colonnes.

J'ai aussi dû modifier init_rotation_euler() pour mat4 qui faisait la multiplication dans l'ordre inverse, a priori pas le bon si j'en crois wiki, (maintenant c'est RzRyRx). test_math renvoie toujours 0 fail, donc soit je ne l'ai pas unit testée, soit je n'ai plus rien à foutre devant un ordinateur.

J'ai un peu honte, et je crois me souvenir avoir bidouillé pareil dans WEngine. Le fait est que ça fonctionne, et que seule la caméra sera "bancale", tous les objets de la scène utilisent *Transformation* et ne sont donc pas affectés par mes modifs.

Quand j'en serai à animer une caméra, soit j'écrirai une nouvelle classe exprès avec support quaternion, soit j'ajouterai des fonctionnalités à *Camera*, pour l'instant j'ai juste besoin d'une caméra FPS.

[BUG][fixed] Specular reflections

Elles sont… fantaisistes. Pas sûr que j'ai envie d'y passer des heures vu que bientôt c'est PBR. Mais y a ptêtre hippopotame sous caillou.

J'ai réglé un problème mais il y en a un autre. Mes réflexions spéculaires ne semblent fonctionner que quand je suis en (0,y,0). Note : mon G-Buffer est en world space, comme le sont mes (mauvais) calculs de lumière.

Bon, réglé pour la lumière directionnelle en envoyant le xz-flip de la position de la cam en uniform oO. x -> -x, y -> y, z -> -z (== rotation de 180° autour de y du vecteur position) J'essaye de biter. En revanche, ça contrarie les spécus pour les point lights, va putain de comprendre. Elles sont orientées correctement selon l'axe gauche-droite de l'écran, mais pas selon l'axe avant-arrière.

Si cependant on envoie la position non modifiée, les spécus des point lights sont mal orientées gauche-droite et correctement avant-arrière. Et dans "lpass.frag", remplacer

```
    vec3 halfwayDir = normalize(lightDir + viewDir);
    // par
    vec3 halfwayDir = normalize(reflect(-lightDir,normal) + viewDir);
```

Règle de manière ad hoc le sens des spécus point light quand on envoie le xz-flip de la position.

Vérifier si les normales sont dans le bon sens

Résolution: voir le 18-07-18

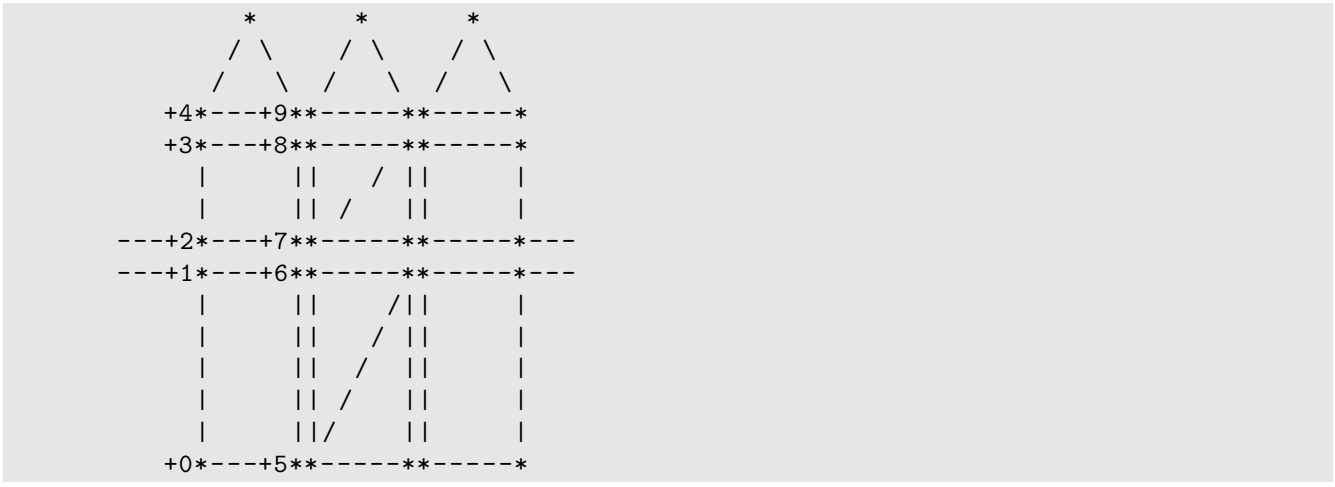
[17-07-18]

Aujourd'hui j'ai laissé de côté les déboires d'hier et me suis consacré à quelques aspects orthogonaux avec le potentiel de me remonter le moral.

[Procedural] Génération procédurale de cristaux

J'ai écrit un algo bête et méchant pour générer des mesh en forme de cristaux oblongs dans mesh_factory. La fonction prend une seed en entrée et crache un mesh de cristal. Ce qui se passe au milieu est un peu plus rébarbatif. Basiquement, un polygone à n côtés (n entre 3 et 7) est généré ainsi que 2 copies rescaled et translatées vers le haut. * Les points du premier polygone sont échantillonnés sur un cercle et perturbés radialement. * Le polygone du dessus est le même en plus grand (plus étant une variable aléatoire entre 1 et 1.5). * Le polygone du haut est le même en plus petit (avec un facteur d'échelle entre 0.5 et 0.8). * La hauteur des deux polygones du dessus est aussi aléatoire. * Il y a un vertex au centre du polygone supérieur, afin de pouvoir trianguler le mesh facilement.

Faut imaginer qu'on recolle et "enroule" les polygones ci-dessous. Il y a bien des vertices répétés, ça permet d'affecter des normales par face, et d'avoir une lumière bien "taillée". Les vertices supérieurs sont à la même position (vertex au centre du polygone supérieur).



Je range les vertices de bas en haut dans le mesh, et le top vertex est le dernier. De fait je peux affecter les triangles facilement : en un vertex marqué "+2" l'indice vaut (10 * F)+2 avec F le numéro de la face (de 0 à n-1).

[FXAA] Fast Approximate Anti-Aliasing, L'algo qui fonctionne, et qui fait pas chier.

De l'anti-aliasing screen-space (post processing) ! Eh ouais ! Un des gros problèmes du deferred shading est la difficulté (et le coût) d'implémentation des méthodes classiques d'anti-aliasing (MSAA: Multi-Sampled AA). Par ailleurs, MSAA et al. sont en général des algos très couteux en temps de calcul.

Une technique concurrente extrêmement rapide a fait son apparition avec des jeux tels que Skyrim, Battlefield 3 et Batman: Arkham City : FXAA. Cette technique est screen space, et analyze chaque pixel et son voisinage afin de déterminer si ce pixel contribue à créer un artéfact d'aliasing (arête artificielle). Si c'est le cas, l'algo lisse le pixel.

J'ai trouvé le code sur le net ici : https://stackoverflow.com/questions/12105330/how-does-this-simple-fxaa-work Un autre mec s'étonnait que l'algo fonctionne aussi bien et demandait des détails sur le fonctionnement. Pas fréquent de pouvoir tester avant de comprendre ! J'ai encapsulé ça dans une fonction qui attend les mêmes arguments que la fonction texture() de glsl, ce qui permet de remplacer

```
    vec3 hdrColor = texture(screenTex, texCoord).rgb;
    // par
    vec3 hdrColor = FXAA(screenTex, texCoord); // Fast Approximate Anti-Aliasing
```

dans le shader de post-processing.

Avantages de l'algo : * Test *tous* les pixels de l'écran, donc même ceux issus de blending ou d'autres fragment shaders (alors que MSAA ne peut pas les "voir" nativement). * Fucking rapide.

Sources:

```
[1] https://blog.codinghorror.com/fast-approximate-anti-aliasing-fxaa/
[2] http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/
FXAA_WhitePaper.pdf
```

J'ai un peu commenté le code et fait quelques optimisation en me basant sur le WhitePaper de Nvidia. Je fais un test local de luminance pour discriminer rapidement les pixels non aliasés et retourner rapidement lorsque c'est le cas. Je fais le test de luminance sur les canaux R et G seulement.

Notes

Mes spécu dirlight ne sont clairement pas dans le bon sens : c'est comme si les rayons étaient renvoyés au lieu d'être réfléchis. J'ai réglé ça en réfléchissant le lightDir comme pour les point-lights :

```
    vec3 halfwayDir = normalize(reflect(-lightDir,normal) + viewDir);  // WTF
        reflect?
```

C'est du provisoire, le temps que je bite. Y a le flag **WTF** à côté pour que je me souvienne.

[Fog]

Du bon gros fog non linéaire, rempompé sur WEngine, modifié pour fonctionner en deferred :

```
    float dist      = length(fragPos - rd.v3_viewPos);
    float fogFactor = 1.0/exp(pow((dist * FogDensity),3));
    fogFactor       = clamp(fogFactor, 0.0, 1.0);
    total_light = mix(fogColor, total_light, fogFactor);
```

C'est appliqué avant le debug overlay et le wireframe, donc on les voit même à travers le fog. **Touche F pour activer/désactiver**

[18-07-18]

J'essaye de repartir sur le bug des spécu. Il semblerait que tout mon repère soit inversé. Par exemple, le haut est le sens des y négatifs !! Nécessairement, ma lumière est calculée à l'envers puisque toute la scène est à l'envers. Je crois que je trimbale une erreur depuis le début dans les quats, et tout à été construit par dessus pour la compenser : matrices de projection et de vue de la cam, matrices modèles des objets, illumination, tout y passe. C'est la hess puissance 1000.

Résolu a priori. La matrice de translation de la matrice modèle (pour la cam uniquement) est maintenant initialisée avec -position_. J'imagine que ça fait sens : (0,0,0)-position_ est le vecteur position du point position_ et y a l'idée qu'on translate le monde et non la caméra en OpenGL… C'est l'ambiguité alias/alibi des rotations.

Les coordonnées du "player" ne sont plus inversées. Je peux toujours me gourrer mais je le sens bien ! En tout cas les calculs de lumière sont maintenant conformes : plus de flag WTF dans mes phucking shaders. Donc je n'ai plus aucune crainte concernant l'implémentation du PBR.

Tout ça a eu un effet assez inattendu : les textures d'Erwin sur les cubes volants devenaient pixelisées à mort sous certains angles d'éclairage. J'ai identifié et réglé un problème d'exposant spéculaire trop bas que j'ai eu hier en faisant mon "fucking around" constructif du soir. Exposant spéculaire à 0 avec un Blinn-Phong ? -> Pixélisation dégueulasse de tout l'écran. Si c'est un comportement défini, ça devrait surement pouvoir être exploité pour un effet de merde.

[ERROR][~fixed] G-Buffer with internal format GL_UNSIGNED_BYTE

Dans *TextureInternal* j'utilise glTexImage2D() avec GL_UNSIGNED_BYTE comme type. Du coup, mon G-Buffer cap à 127. C'est pour ça que j'ai des emmerdes avec l'exposant spéculaire.

En fait ça ne change pas grand chose de passer le G-Buffer en GL_FLOAT. On étend le range entre -127.0 et 127.0, certes, par flemme j'utilise seulement l'intervalle positif pour encoder l'exposant spécu. Aussi, je fait un calcul de luminance dans lpass pour calculer l'intensité spéculaire, le résultat est assez joli ! Mais bon -> PBR -> osef -> …

[Deferred] +Forward

On a une passe forward qui fonctionne au dessus de la passe deferred pour permettre le blending plus tard. Pour l'instant le shader "forwardstage" sort une couleur unie. La passe forward écrit sur la texture "screen".

La *Scene* contient provisoirement un vector de shared_ptr sur *Model* pour les modèles qui doivent passer par le forward stage. J'ai poussé un cristal là-dedans, il apparaît en rouge uniforme dans la scène, comme prévu.

IMPORTANT *Scene* permettra à l'avenir d'itérer sur des *Mesh* plutôt que sur des *Model*. En effet, un modèle peut en pratique contenir des mesh avec ou sans transparence, parfois l'une et l'autre (imaginer un modèle de casque avec une visière translucide…).

Comme le depth buffer obtenu lors de la passe géométrique n'est pas accessible lors de la forward pass, si l'on se contente de dessiner dans le *ScreenBuffer* on ne verra soit rien, soit un modèle affiché par dessus tout le reste, selon l'état de GL. Le truc malin à faire c'est de bliter (copier) le depth buffer du G-Buffer dans le FBO du screen buffer. Comme ça on restaure le contexte de profondeur de la passe géométrique, et on peut depth tester en dessinant les objets de la passe forward. J'ai écrit une fonction pour bliter le depth buffer d'un *FrameBuffer* à l'autre et d'un *ProcessingStage* à l'autre :

```
void FrameBuffer::blit_depth(FrameBuffer& destination) const
{
    // write depth buffer to destination framebuffer
    glBindFramebuffer(GL_READ_FRAMEBUFFER, frame_buffer_);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, destination.frame_buffer_);
    glBlitFramebuffer(0,            // src x0
                      0,            // src y0
                      width_,       // src x1
                      height_,      // src y1
                      0,            // dst x0
                      0,            // dst y0
                      destination.width_,   // dst x1
                      destination.height_,  // dst y1
                      GL_DEPTH_BUFFER_BIT,  // mask
                      GL_NEAREST);  // filter
}
```

Du coup, dans le renderer (main) on fait :

```
    gbuffer.blit_depth(sbuffer);
    forward_stage_shader.use();
    blend_vertex_array_.bind();
    sbuffer.draw_to([&](){ /*...*/ });
    // ...
```

[TODO] Améliorations

```
[o] Plutôt que de passer un bool seul pour le type sous-jacent des textures,
   passer un vecteur de bool. Les layers couleur n'ont rien à foutre en float.
    -> J'ai viré le bool, je décide du type en fonction du format interne dans le
       constructeur de texture. Plus facile.
[ ] Ce sont les _Mesh_ qu'on devrait "traverse" dans _Scene_, le foncteur
   prendrait un argument sup avec un pointeur sur _Material_. La scène associe
   mesh et material à partir de l'information contenue dans _Model_.
[x] Définir des textures par défaut pour simplifier la création d'assets. Un asset
   ne définit pas de normal map ? Pas grave, y a une normal map par défaut ! etc.
```

[19-07-18]

- Des textures par défaut sont maintenant chargées quand un asset ne les utilise pas toutes (le moteur, lui, en a besoin).
- Alpha blending de base pour les mesh de la scène qui vont dans la forward pass.
- *GBuffer* possède une nouvelle texture “propsTex” qui pour l’instant ne contient que l’exposant spéculaire (shininess) dans le canal rouge. On y mettra la métallicité, par exemple.

[Normal Mapping]

J’ai implémenté un début de normal mapping. Les meshs de la scène utilisent maintenant le format de vertex *Vertex3P3N3T2U* qui inclue un attribut pour les vecteurs tangents. Ces tangentes sont calculées en même temps que les normales si l’on utilise la fonction Mesh::build_normals_and_tangents(), et sont lissées en même temps que les normales si l’on utilise la fonction Mesh::smooth_normals_and_tangents().

Une normal map (ou bump map) est définie dans l’espace tangent (au triangle à texturer). C’est pourquoi la teinte générale des normal maps est bleue : car les normales sont en moyenne orientées dans le sens de l’axe z (= canal bleu en RGB), hors de l’écran. Donc il faut transformer les normales samplées dans la normal map de l’espace tangent vers l’espace monde si l’on veut faire les calculs de lumière corrects. C’est le rôle de la matrice TBN (Tangent, Bitangent, Normal) calculée dans gpass.vert, qui convertit un vecteur de l’espace tangent vers l’espace monde :

```
    vec3 N = normalize(tr.m3_Normal * in_normal);
    vec3 T = normalize(tr.m3_Normal * in_tangent);
    vec3 B = cross(N, T);
    mat3 TBN = mat3(T,B,N);
```

Noter que la bitangente (je suis immature, ce nom me fait marrer), est calculée sur le tas, par produit vectoriel de la normale et de la tangente données en attributs.

C’est le fragment shader qui fait la multiplication matricielle pour convertir les normales samplées :

```
    vec3 normal = texture(mt.normalTex, frag_texCoord).rgb;
    normal = normalize(normal*2.0 - 1.0);
    out_normal = normalize(frag_TBN*normal);
```

Ce n'est *PAS* économe : le fragment shader est exécuté beaucoup plus souvent que le vertex shader. C'est pourquoi les gens en général transposent cette matrice dans le vertex shader pour l'inverser, et passent au fragment shader la position du fragment, de la caméra et des lumières *dans l'espace tangent* (les vecteurs concernés sont multipliés par l'inverse de TBN), et les normales elles, sont déjà dans le bon espace.

Le problème que ça me pose est que je travaille en différé, donc les positions de la caméra et des lumières qui ne sont pertinentes que lors de la lighting pass… Ben elles restent dans l'espace monde, parce que je n'ai plus de matrice TBN sous la main hors de la passe géométrique. Bon, après, sur un thread de 2006 un gars raconte que c'est pas hyper important (même sur le matos de l'époque) : https://social.msdn.microsoft.com/forums/en-US/d7b39b2f-6966-4302-a06d-6d0c97bf698e/question-on-deferred-shading-with-normal-mapping

J'ai trouvé cet outil fantastique pour générer des normal maps (et autres textures du genre) à partir de photos ou d'une height map : http://cpetry.github.io/NormalMap-Online/ **Juste penser à inverser R et G**

Notes

- Les tangentes lissées désorthogonalisent la matrice TBN, on peut utiliser le *processus de Gram-Schmidt* pour la réorthogonaliser :

```
vec3 N   = normalize(tr.m3_Normal * in_normal);
vec3 T = normalize(tr.m3_Normal * in_tangent);
// re-orthogonalize T with respect to N (Gram-Schmidt process)
T = normalize(T - dot(T, N)*N);
vec3 B = cross(N, T);
vertex_TBN = mat3(T,B,N);
```

- J'ai clairement un début de bourrage dans la pipeline, certaines frames buffswap pendant environ 7ms (activité graphique à ~ 45% de la frame) en 1920x1080, donc va falloir que j'optimise à donf le G-Buffer. C'est clairement lié à ça, en basse résolution (petite fenêtre 1024x768) je n'ai pas ce problème. Ce sont les grosses textures de la taille de l'écran qui bouffent.
- Pour le rendu des objets transparents avec alpha blending, il est nécessaire de les trier back to front avant de les rendre. *Les objets transparents les plus distants doivent être rendus en premier* ! **OU BIEN** implémenter un *stencil routed A-Buffer* (Alpha-Buffer) pour de l'*order independent transparency*.
- Pour les shadow maps, plus tard, un avantage de l'approche différée multipasse est que je peux utiliser une seule texture shadow map et la réécrire du point de vue de la lumière courante. Le shader lpass sera exécuté une fois pour chaque lumière, et les résultats de toutes les passes sont combinées par additive blending dans le L-Buffer (mon *ScreenBuffer*).
  - Pour rendre les shadow maps en fonction du type de lumière :
    * Directional : orthographic projection
    * Spot : perspective projection
    * Point : omnidirectional map -> cube map (=6 textures par lumière ponctuelle) *OU BIEN -> Dual Paraboloid Shadow Mapping* où on n'utilise que 2 textures par lumière en mappant 2 hémisphères autour de la source. source : Shadow Mapping for Hemispherical and Omnidirectional Light Sources (Stefan Brabec et al. 2002)
      https://pdfs.semanticscholar.org/eb42 /f9330b6cdb1a0b12595a33c2674e0fdaea12.pdf

[GBuffer] Optimisation : compression des normales

L'article Inferred Lighting: Fast dynamic lighting and shadows for opaque and translucent objects (Kircher et al.) http://www.students.science.uu.nl/~3220516/advancedgraphics/papers/inferred_lighting.pdf Donne une transformation hyper conne à effectuer sur les normales avant stockage dans le G-Buffer, de manière à gagner une composante !

```
direct  : n -> n'=normalize(n+vec3(0,0,1)).xy
inverse : z = sqrt(1-pow(length(n'),2)) ; n' -> (2*z*n'_x, 2*z*n'_y, 2*pow(z,2)-1)
```

J'ai tout simplement écrit les fonctions compress_normal() resp. decompress_normal() dans gpass.frag resp. lpass.frag, et je viens de gagner un byte par pixel dans le G-Buffer.

Cet thèse : Real-time Lighting Effects using Deferred Shading (Michal Ferko, 2012)

```
http://www.sccg.sk/ferko/dpMichalFerko2012.pdf
```

Explique également que les half-floats c'est pas suffisant en terme de précision pour des normales. Il vaut mieux utiliser les formats internes en *SNORM* d'OpenGL (16 bits unsigned). J'ai changé le format de normalTex en *GL_RG16_SNORM*.

A priori le format SNORM est aussi intéressant pour stocker du bruit (Perlin, Simplex)… **Mais pas sûr que ce soit core**

[TODO] Améliorations et suite du programme

```
[x] Optimiser à donf le G-Buffer.
    -> Du mieux que j'ai pu pour l'instant.
[x] Faire du parallax mapping.
    [ ] Une fois les textures height maps introduites, générer automatiquement les
       normal maps à partir d'un filtrage (Sobel ?) sur les height maps.
        [ ] On pourrait même utiliser un shader pour faire ça...
[x] Trier la scène front to back avant la passe géométrique. Les fragments les
   plus proches étant dessinés en premier, il est de plus en plus probable que les
   fragments suivants échouent au depth test et soient rejetés. Donc on élimine un
   grand nombre de potentiels rapidement, et la passe est exécutée plus rapidement.
    [x] Les objets transparents eux sont triés back to front.
[o] Condenser les screen quads draws dans un seul appel statique plutôt que de
   définir et initialiser un _BufferUnit_ et un _VerterArray_ à chaque fois qu'on
   veut rendre un quad.
    ->[23-07-18] Pas hyper compatible avec renderer.hpp...
```

lire : http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/

[Command Line Galore]

Compter les lignes de code en récursif depuis le dossier courant :

```
>> find . -name '*.cpp' | xargs wc -l
```

[Quotes] de porc

- Reconstruire la position en view space depuis le G-Buffer. Ferko : "Thanks to the depth buffer, the other large texture storing pixel positions becomes unnecessary. During the G-buffer generation, we have to use a depth buffer, and thankfully, the depth value can be used to reconstruct pixel position. We take pixel coordinates $(x, y) \in [0, 1]^2$ and depth $z \in [0, 1]$ and transform these back into clip space by mapping the target range into $[-1, 1]^3$. We then apply the inverse of the projection matrix used when generating the G-Buffer. The result is the point's view-space position, which is then used as the position."
- Mon S-Buffer est en fait un L-Buffer : Ferko : "If we intend to perform only lighting calculation using additive blending, we can render directly into the window's framebuffer. However, there are still many post-processing effects that can be applied after the lighting, and therefore *it is useful to store the illuminated scene into a texture*. This texture is *called the L-Buffer* or the lighting buffer. This intermediate step becomes necessary when using High Dynamic Range Lighting, which will be further discussed in Section 4.1." -> Renomer *ScreenBuffer* en *LBuffer*.

[Lighting] Nouvelle fonction d'atténuation

```
float attenuate(float distance, float radius)
{
    // originally pow(,20) but I found it fucks up specular reflection tails
    return (1.0 - pow(distance/radius, 5))/(1.0 + distance*distance);
}
```

au lieu de :

```
    float attenuation = 1.0 / (pl[ii].v3_attenuation.x + pl[ii].v3_attenuation.y *
       distance +
                          pl[ii].v3_attenuation.z * (distance * distance));
```

-> Plus besoin de pl[ii].v3_attenuation : on a juste besoin du radius ! -> Plus de ces affreux cercles à distance des point lights, on a un falloff en douceur !! -> Mais consomme de l'intensité dans les spéculaires.

Je suis pas hyper satisfait de la fonction en soit, *faudra tweeker*, mais l'approche consistant à passer un rayon de sphère tout seul pour calculer l'atténuation est bien sympa.

[20-07-18]

[Parallax Mapping]

Hop, ça c'est fait.

Un *Material* doit être configuré pour pouvoir être rendu avec *normal mapping* et *parallax displacement mapping*. Il faut appeler la méthode set_normal_map(true) pour activer cette option de rendu et set_parallax_height_scale(0.1f) pour (par exemple) définir une profondeur d'extrusion à 0.1f. Ceci fait, la passe de géométrie se charge de déplacer les coordonnées de texture en fonction de la direction de vue, afin de simuler la parallaxe et ainsi donner l'illusion que la texture n'est pas plate. Bien sûr, l'asset doit contenir les textures *assetName_mt.normalTex.png* et *assetName_mt.depthTex.png*. Ces textures sont optionnelles et donc aucune texture par défaut n'est prévue.

La fonction parallax_map() du shader gpass.frag calcule les nouvelles coordonnées de texture itérativement. L'algo est adaptatif et estime linéairement le paramètre de marche du rayon en fonction de l'angle de vue. Une étape d'interpolation entre les coordonnées de texture avant et après intersection donne une estimation finale relativement fiable (*parallax occlusion mapping*).

Sources:

```
[1] http://sunandblackcat.com/tipFullView.php?topicid=28
[2] https://learnopengl.com/Advanced-Lighting/Parallax-Mapping
```

[HACK]

Je dois inverser l'axe y de viewDir dans gpass.frag pour que la perspective fonctionne dans le bon sens sur le sol le long de l'axe z. Sinon elle est inversée, ce qui donne une sale impression de merde. Faut que je bite ce qui se passe.

[Optim]

Je bosse sur l'optimisation et les outils de profiling. La classe *MovingAverage* me permet de mesurer le temps de rendu moyen avec écart type. En full HD on est à 8-9ms ce qui est trop à mon avis. Donc j'aimerais préparer le terrain pour du frustum culling dès maintenant, gros morceau nécessaire. Donc il me faut une classe *AABB* et un moyen de les afficher.

Je pense également que la passe géométrique prend trop de temps. Avec une scène triée pour éviter les appels inutiles du fragment shader ça devrait aller mieux.

Note sur la mesure du temps de rendu :

Il est capital d'appeler *glFinish()* une fois avant le start timer et une autre fois avant le stop timer, pour mesurer correctement le temps de rendu. Ca attend qu'OpenGL termine les calculs côté GPU.

Debug geometry

Les géométries de debug (l'affichage d'AABB, OBB etc.) seront poussées séparément des objets 3D lors de la passe géométrique. Elles seront rendus avec GL_LINES.

Fait.

[AABB] Axis Aligned Bounding Boxes

Il aura fallu que je me creuse un peu la tronche.

Nouvelle classe *AABB* que possède chaque modèle. Lors de l'appel à Model::get_AABB(), le AABB est lazy initialisé (ses vertices, et sa transformation (pour l'affichage)). Le calcul d'un AABB est simplifié si l'on a un OBB (*Oriented Bounding Box*). Ca c'est facile, il suffit de tracker la *Transformation* du *Model* (à l'initialisation de l'*AABB* ce dernier reçoit une référence vers la *Transformation* du *Model* parent). Cette transformation multipliée par un offset de position paramétrable et l'échelle intrinsèque du *Model*, forme la *transformation propre* de l'OBB. L'échelle intrinsèque est calculée depuis les dimensions intrinsèques du modèle, elles même calculées à l'initialisation du modèle (coordonnées extrémales en model space).

En appliquant la transformation propre de l'OBB aux vertices d'un cube normalisé en model space (centré en 0), on obtient les *vertices de l'OBB* en world space. Il s'agit ensuite de calculer les coordonnées extrémales de cet ensemble de vertices. On a donc des valeurs minimales en maximales pour x, y et z. Alors les vecteurs :

```
s = (xmax-xmin, ymax-ymin, zmax-zmin)
p = (xmax+xmin, ymax+ymin, zmax+zmin)/2
```

sont respectivement la diagonale de la matrice d'échelle de l'AABB et la position world space du centre de l'AABB (coïncident avec 0 en model space). On peut alors calculer les matrices d'échelle et de position, respectivement Ms et Mp.

La transformation de l'AABB est alors le produit matriciel :

```
offset*Mp*Ms
```

que l'on peut alors appliquer à chaque vertex d'un cube normalisé de l'espace modèle afin d'obtenir les *vertices de l'AABB*. Boom !

Les bounding boxes ne sont pas affichées par défaut, il faut presser "b" pour activer leur rendu (**DEBOUNCER CES PUTAINS DE TOUCHES**). Elles sont générées à la volée dans la passe forward pour l'instant.

Les *AABB* vont me servir principalement pour le *frustum culling*. Mais c'est certain que d'autres techniques pourront en profiter. Donc bon boulot.

Optimisation possible : ne calculer les AABB qu'une seule fois pour les modèles statiques !

[22-07-18] Refactoring

Grosse réorganisation du code. Les énormes sections de rendu dans main() ont été encapsulées dans des classes héritant de l'interface *Renderer*. Nous avons donc :

- *GeometryRenderer* : public Renderer
- *ForwardRenderer* : public Renderer
- *LightingRenderer* : public Renderer

- *BloomRenderer* : public Renderer
- *DebugRenderer* : public Renderer
- *PostProcessingRenderer* : public Renderer

Au passage, *BlurPass* est devenue *BloomRenderer*.

ScreenBuffer est devenu *LBuffer* et la classe parent *ProcessingStage* est devenue *BufferModule* et a été vidée de ses objets reliés à l'envoi de géométrie. Les objets héritant de *BufferModule* tels que *GBuffer* et *LBuffer* sont pensés comme des textures multi-cibles que l'on peut bind comme source ou bien comme cible. Ces objets regroupent une *Texture* et un *FrameBuffer* sous le capot afin d'assurer ces fonctionnalités, mais elles sont clairement orientées données maintenant.

Ce sont les renderers qui doivent pousser la géométrie et la dessiner, aucune autre classe. Dans la liste donnée précédemment, on observe 2 types de renderers, ceux qui travaillent avec des positions (*Vertex3P*) uniquement sont a priori ceux qui travaillent en screen space (*quad renderers*), et ceux qui gèrent des formats de vertex plus complets (*Vertex3P3N3T2U*) qui dessinent des *Mesh* de la *Scene*. J'ai défini une spécialisation pour les quad renderers de sorte qu'ils chargent un unique quad screen space lors de l'appel à load_geometry().

Les renderers possèdent tous un *BufferUnit* et un *VertexArray*, mais pas de *Shader* par défaut, ce sont les classes enfant qui les définissent et les initialisent, je veux avoir un peu de liberté avec ça. Tous les renderers sont initialisés avec une référence vers la *Scene*. Cette référence est non const pour l'instant, parce que j'ai laissé un problème chiant polluer ma codebase : Quaternion::get_model_matrix() mute le quat en le renormalisant et est donc non const, ça a contaminé toute la call stack !

Réglé ce problème de merde en faisant un truc dégueu mais qui a du sens vu de loin :

```
mat4 Quaternion::get_rotation_matrix() const
{
    const_cast<vec4*>(&value_)->normalize();
    // ...
```

Fuck you. La référence vers la *Scene* restera non const cependant, il est vrai que les *BufferUnits* modifient les *Mesh* avec set_buffer_offset(). Mais j'ai quand même un peu fait le ménage dans ma non-const-iness.

[InputHandler]

InputHandler est une nouvelle classe qui me sert pour l'instant de niveau d'indirection supplémentaire pour les keybindings et la gestion de la souris (pour pouvoir faire du remapping plus tard). Cette classe possède plusieurs registres pour les keybindings et les éventuels temps de cooldown (pour le key debouncing). On s'en sert comme ça :

```
    input_handler.stroke_debounce(window, H_("k_ascend"), [&]()
    {
        pcam->ascend(dt*speed_modifier);
    });
```

Donc encore un lambda dégueulasse, qui n'est exécuté que si la touche correspondante au key binding "k_ascend" est dans un état cible donné (pressée, relâchée), et que le timer interne de debouncing est à 0 (cooldown). C'est l'appel à stroke_debounce qui décrémente le timer correspondant ssi l'action n'a pas été exécutée. Le timer est réinitialisé à une valeur propre au keybinding si l'action a été exécutée.

[Frustum Culling]

Done.

On voit demain pour les détails, j'suis crevé.

[Serialize] Flatbuffers

Je veux pouvoir sérialiser / désérialiser facilement certains objets, dont la position de la caméra. Le but immédiat est de rendre répétables certains tests in game. Plutôt que de faire du ad hoc, comme je vais devoir sérialiser tout un tas de trucs à l'avenir, je me suis penché sur une solution plus générique : *Flatbuffers* qui est un outil open-source de génération automatique de classes de sérialisation, multi-langages milti-OS. L'alternative principale est *Google Protocol Buffers*.

Pour compiler le compilateur *flatc* qui permet de transformer un *schema file* en classe de sérialisation, j'ai fait ceci :

```
>> git clone https://github.com/google/flatbuffers.git
>> cd flatbuffers/
>> mkdir build;cd build
>> CC=/usr/bin/clang-6.0 CXX=/usr/bin/clang++-6.0 cmake .. -G "Unix Makefiles"
>> make
>> sudo make install
>> sudo cp ./flat* /usr/bin/
```

Le makefile génère les exécutables *flatc*, *flathash*, *flatsamplebinary*, *flatsampletext* et *flattests*. Le make install n'installe que les headers, donc il faut copier manuellement les binaires quelque part dans le PATH.

flatc compile des schema files écrits en dans un IDL (*Interface Definition Language*) fortement typé C-like.

[23-07-18]

[Optim]

[Scene sorting]

J'ai modifié les fonction traverse_() de la classe *Scene*. Elles possèdent toutes un prédicat avec une valeur par défaut en second argument (plus de traverse_models_if(), juste traverse_models() avec un deuxième argument pour la clause if). De plus, je me livre à un petit tour d'optimisation qui consiste à trier la scène d'avant en arrière, de sorte qu'un maximum de fragments ne passent pas le depth test. Pour cela j'initialise une liste avec des indices dans l'ordre arithmétique, avec autant d'éléments qu'il y a de modèles, puis je trie cette liste avec std::sort et un comparateur custom en lambda, qui évalue et compare les distances des objets par rapport à la caméra. Ainsi j'obtiens la liste des permutations dans un vecteur membre de la *Scene*, et lors d'un traverse, je peux parcourir le vecteur de modèles dans l'ordre permuté.

J'en ai profité pour implémenter la même mécanique afin de trier les objets transparents d'arrière en avant et ainsi avoir un blending correct.

Les fonctions sort_models() et sort_transparent_models() sont appelées dans le Scene::update(). On pourra veiller à mettre en cache les vecteurs de permutations plus tard. Pour traverser la scène en utilisant les permutations, j'ai écrit les fonctions traverse_models_sorted_fb() et traverse_models_blend_sorted_bf(), mais peut être que je vais homogénéiser ça en rajoutant un troisième argument à traverse_models().

[Frustum Culling]

Pour réaliser le frustum culling, je dois dans un premier temps calculer les positions des points du frustum en world space. Je m'aide pour cela des données de l'objet *Frustum* de la camera et de quelques relations de Chasles.

Pour commencer, on extrait de la caméra la matrice de vue V et le frustum F(l,r,b,t,n,f). On a donc :

```
hnear = h = t-b
wnear = w = r-l
ratio = w/h
hfar  = h/n * f
wfar  = ratio * hfar
```

De la view matrix on peut extraire les vecteurs :

```
_right   = V.row(0)
_up      = V.row(1)
_forward = V.row(2)
_p       = -V.row(3)
```

(ou utiliser get_position() pour p)

On peut donc calculer les centres des plans near et far :

```
_nc = _p + _forward * n
_fc = _p + _forward * f
```

Les points du frustum (RBN, RBF, LBF, LBN, RTN, RTF, LTF et LTN) sont calculés par addition vectorielle (faire un schéma aide beaucoup) :

```
RBN = _nc - (_up * hnear/2) + (_right * wnear/2)
RBF = _fc - (_up * hfar/2)  + (_right * wfar/2)
LBF = ...
```

Ensuite on calcule les normales à chaque plan :

```
normal_L = cross(LBF-LBN, LTN-LBN)
normal_R = cross(RTN-RBN, RBF-RBN)
normal_B = ...
```

Pour normal_L par exemple, on s'est servi du point LBN comme point commun, donc la donnée de ce point LBN et de la normale au plan gauche normal_L suffit à encoder l'information sur le plan de gauche. Donc on mémorise ces points en les associant à chaque normale, c'est important pour la suite. Noter que les normales pointent *vers l'intérieur* du frustum. Rien de particulier, mais c'est une convention à laquelle il faudra se tenir.

Mettons maintenant qu'on veuille tester si un point P est *au dessus* d'un plan défini par un point P0 et une normale n. La notion de dessus découle de l'orientation du plan, et on dira que le point P est au dessus si on a :

```
(_P - _P0) . _n > 0
```

Par conséquent, en suivant nos conventions, un point P est dans notre frustum s'il est au dessus de tous les plans du frustum.

Pour le test de collision frustum / AABB, je décide d'exclure un AABB (l'objet sera culled) ssi tous ses points sont en dessous du **même** plan. Ca permet d'éviter tout un tas d'emmerdes quand ancun vertex de l'AABB n'est dans le frustum mais qu'au moins une de ses arêtes l'est (ce qui fait que l'objet peut être visible), ce qui est hyper probable avec les mesh étendus comme les terrains.

[**HACK**]

Problème de compatibilité entre les mesh cubiques centrées sur la face B ou au centre du cube -> J'ai du magouiller dans le calcul des AABB pour appliquer une transformation légèrement différente de la transformation propre (à une translation verticale près) aux vertices d'un cube en espace modèle, sans quoi les AABB sont trop hauts ou bien leurs représentations trop basses (rien de grave pour les représentations). Le hack consiste à modifier un élément de la matrice de transformation propre avant de retourner, donc pas cher, mais inélégant. Voir bounding_boxes.cpp.

[Optim] Light Volumes

Mon test de rayon des sources lumineuses avec son gros if dans lpass.frag commence à faire pitié (les conditions de branchement dynamiques dans les shaders pètent le *front d'onde* et sont donc attrocement sous-optimales). La bonne façon de faire est de rendre effectivement des meshs de sphères, ainsi les fragments ne seront évalués qu'à l'intérieur des sphères. Ce faisant, on élimine plein de problèmes, dont celui du nombre de lumières à définir dans le shader. Non, là on fait un draw call par light volume, donc le shader ne voit toujours qu'une seule lumière à la fois.

Tout ça permet une approche assez unifiée et découplée : * On pourra **cull les lumières** comme de la vulgaire géométrie * donc prévoir un test de collision sphère / frustum * Pour chaque type de lumière on a une géométrie différente : * le *quad* écran tout entier pour la lumière directionnelle * une *sphère* pour une lumière ponctuelle omnidirectionnelle * un *cône* pour un spot lumineux * Donc les seuls branchements conditionnels seront uniformes et donc prédictibles par le GPU. [x] Déjà il faut rendre des sphères.

Pour le blend des passes light volume :

```
    glEnable(GL_BLEND);
    glBlendEquation(GL_FUNC_ADD);
    glBlendFunc(GL_ONE, GL_ONE);
```

Good, je peux rendre des sphères. Pour générer un *Mesh* de sphère, appeler factory::make_sphere_3P(n_rings, n_points_on_ring). J'ai modifié le debug renderer pour qu'il puisse afficher les light volumes en wireframe.

Tout d'abord, j'ai modifié la classe *Light* pour lui rajouter une méthode virtuelle get_model_matrix(). Une lumière ponctuelle renvoie une matrice de translation multipliée à droite par une matrice d'échelle (l'échelle étant son rayon).

Le *DebugRenderer* peut afficher les light volumes en traversant les lumières de la scène et et récupérant leur matrice modèle avec laquelle on forme la matrice MVP qui est envoyée au line_shader. Le renderer dessine alors une sphère (dont la géométrie est poussée en amont par load_geometry()). Il faut appuyer sur **L** pour activer le rendu des light volumes.

[Zik]

Chrome Canyon - Elemental Themes Nom de Dieu !!!

[24-07-18]

[Bloom]

J'ai modifié *BloomRenderer* pour ne sortir qu'une seule texture nommée “bloom” et associée au sampler “bloomTex”. Cette texture est obtenue lors de la passe verticale par blending itératif. Les coefficients de blending sont GL_SRC_ALPHA et GL_ONE_MINUS_SRC_ALPHA, l'alpha est passé en uniform au shader, ce qui me permet de garder le contrôle sur la combinaison des différents niveaux de flou. Ca ne semble pas impacter négativement les performances (4 passes sur une texture de la taille de l'écran contre 4 passes sur des textures de tailles décroissante en puissance de 2 précédemment), j'imagine que le fait de changer de cible de rendu 5 fois contre 8 précédemment doit jouer en ma faveur. Moins de texture lookup en post processing également (une seule texture bloom contre 4 précédemment).

Camera

La classe *Camera* possède maintenant son propre *FrustumBox*. La scène update le frustum de la caméra avec Camera::update_frustum() et les renderers peuvent cull grâce à la fonction Camera::frustum_collides(const AABB&).

[25-07-18]

Je bosse donc depuis 3 jours sur le rendu des *light volumes* en utilisant des *géométries proxy* pour mes sources lumineuses, et el famoso *stencil buffer trick* pour résoudre les problèmes connus d'*overshading* et de disparition des lumières par front culling quand la caméra est dans leurs light volumes. Le rendu des lumières par cette méthode alternative est accessible en activant l'option **EXPERIMENTAL_LIGHT_VOLUMES**. Je retarde l'échéance depuis bien 3h pour ne pas écrire que je suis déçu. J'essaye de retourner le truc dans tous les sens, même si ça fonctionne, j'ai des artéfacts à la con sur le bord des objets quand je bouge, sous certaines conditions d'éclairage. De plus, c'est plus lent sur ma scène actuelle que le if dynamique dégueulasse dans le fragment shader. Probablement

qu'en intérieur avec de l'occlusion culling ça peut être rentable, si on a plein de lumières avec une portée relativement limitée… Si j'arrive à régler les artéfacts de merde…

[Deferred Rendering] Stencil Optimization

Le principe est de rendre les volumes en 2 passes successives, dans la même boucle. Lors de la première : * on désactive l'écriture dans les color buffers avec glDrawBuffer(0) * on clear le stencil * on active le depth test en mode GL_LESS * on désactive le face culling afin que les polygones Front *et* Back soient pris en compte (*two-sided stencil*) * on configure le stencil pour : * que le stencil test passe toujours (GL_ALWAYS) * avec une référence à 0 * on configure le stencil operator pour : * incrémenter la valeur locale quand un back face polygon échoue au depth test * décrémenter la valeur locale quand un front face polygon échoue au depth test * conserver la valeur dans tous les autres cas * on active un *null shader* (passthrough vertex shader qui ne fait qu'une transformation de gl_Position avec la matrice modèle de la lumière courante passée en uniform, et un *fragment shader vide*). Tout ce qu'on veut c'est remplir le stencil. * on dessine le light volume de la lumière courante (draw call)

Explication. Dessiner la sphère sous depth test va cull tous les fragments masqués par la géométrie ambiante. Les fragments dans le light volume sont les seuls qui auront un compte non nul dans le stencil. En effet, seuls les fragments dans le volume vont causer un depth fail en back face (et pas en front). Donc dans la seconde passe, on va simplement se servir de cette information pour que tous les fragments screen-space qui ont une valeur de stencil égale à 0 ne soient pas dessinés (stencil fail).

Alternativement on peut configurer une opération non séparée pour le stencil operator : inversion bitwise de la valeur en cas de depth fail, rien sinon. De manière équivalente seuls les fragments avec une valeur non nulle sont ceux illuminés par la lumière courante. J'ai laissé cette alternative en option, activable en définissant **OPTIM_LIGHT_VOLUMES_STENCIL_INVERT**.

La deuxième passe : * on restaure l'écriture dans les color buffers * on change le stencil test pour passer quand la valeur est différente de 0 * on désactive le depth testing * on active le blending en GL_ONE, GL_ONE (classique pour la lumière) * on active le front face culling * là seulement on active notre shader d'illumination (lpass_exp.frag), on lui envoie tout le bordel habituel * on dessine le light volume une seconde fois

Donc là c'est facile, on n'illumine que là où il faut. Le cull front c'est parce qu'on a juste pas besoin de dessiner les polygones front. Bien sûr on fait de l'additive blending pour obtenir une image avec l'influence composite de toutes les lumières.

Pour la lumière directionnelle (noter que j'ai pas envie de m'emmerder avec plusieurs lumières directionnelles), comme son volume est infini et concerne donc tout l'écran, on n'a pas besoin du stencil, et on la passe séparément avant ou après les autres lumières.

Trois **GROS** avantages de cette approche, et c'est là que ça me fait chier de ne pas en penser que du bien : * On passe les lumières une par une, donc le shader n'en voit toujours qu'une, donc plus besoin de modifier le shader quand le nombre de lumières change… * On peut utiliser une seule shadow map (quand on en sera là) pour toutes les lumières non directionnelles qui projettent des ombres. * On peut cull les light volumes comme n'importe quelle géométrie (j'ai implémenté un algo de collision sphère/frustum, en passant).

Sources:

```
[1] https://kayru.org/articles/deferred-stencil/
[2] http://ogldev.atspace.co.uk/www/tutorial35/tutorial35.html
[3] http://ogldev.atspace.co.uk/www/tutorial36/tutorial36.html
[4] http://ogldev.atspace.co.uk/www/tutorial37/tutorial37.html
[5] https://dqlin.xyz/tech/2018/01/02/stencil/
[6] https://learnopengl.com/Advanced-Lighting/Deferred-Shading
[7]
   http://jahej.com/alt/2011_08_08_stencil-buffer-optimisation-for-deferred-lights.html
```

[Hue Shift] Juste au cas où

```
vec3 hueShift( vec3 color, float hueAdjust ){

    const vec3  kRGBToYPrime = vec3 (0.299, 0.587, 0.114);
    const vec3  kRGBToI      = vec3 (0.596, -0.275, -0.321);
    const vec3  kRGBToQ      = vec3 (0.212, -0.523, 0.311);

    const vec3  kYIQToR     = vec3 (1.0, 0.956, 0.621);
    const vec3  kYIQToG     = vec3 (1.0, -0.272, -0.647);
    const vec3  kYIQToB     = vec3 (1.0, -1.107, 1.704);

    float   YPrime  = dot (color, kRGBToYPrime);
    float   I       = dot (color, kRGBToI);
    float   Q       = dot (color, kRGBToQ);
    float   hue     = atan (Q, I);
    float   chroma  = sqrt (I * I + Q * Q);

    hue += hueAdjust;

    Q = chroma * sin (hue);
    I = chroma * cos (hue);

    vec3    yIQ   = vec3 (YPrime, I, Q);

    return vec3( dot (yIQ, kYIQToR), dot (yIQ, kYIQToG), dot (yIQ, kYIQToB) );

}
```

Trouvé ici : https://gist.github.com/mairod/a75e7b44f68110e1576d77419d608786

[27-07-18]

Rien d'extrême aujourd'hui : du rendu de texte.

[Text Rendering]

La classe *TextRenderer* utilise Freetype 2 pour rasteriser des polices ttf (dans le dossier res/fonts), générer les textures OpenGL pour les 128 premiers caractères de la table ASCII et sauvegarder les références vers ces textures dans une grosse structure associative. Plusieurs fonts peuvent être rasterisées et sont référencées par le hash du nom de fichier sans le ".ttf". La fonction TextRenderer::set_face(hash_t) permet de changer la police courante. Par défaut, la police courante est la dernière police chargée. Il est possible de rendre une ligne de texte immédiatement grâce à la fonction TextRenderer::render_line(…), mais il est aussi possible de mettre une ligne de texte en attente dans une FIFO interne avec TextRenderer::schedule_for_drawing(…), et celle-ci sera dessinée en batch avec toutes les autres lignes qui auront été soumises lors de l'appel à TextRenderer::render() (qui en outre vide la queue au fur et à mesure).

TextRenderer est un *Renderer*. Sa fonction init_geometry() pousse dans son *BufferUnit* un quad de taille 1x1 qui sera dessiné pour chaque caractère de chaque ligne. Pour chaque caractère, une matrice de translation/scaling est calculée à la volée et envoyée en uniform au shader (text.vert). J'avais le choix avec le vertex streaming (ce dont *BufferUnit* est nouvellement capable avec les fonction upload_dynamic() et stream()), mais j'ai jugé plus économe de faire comme ça pour limiter les échanges client-serveur OpenGL. *Ca reste à mesurer.* Mais c'est un bon client pour *l'instanciation* de fait.

[Light Volumes]

J'ai apporté quelques améliorations au rendu par light volumes. * Déjà, je n'envoie plus de sphères lors de la light pass, mais le quad écran tout entier, et je repose sur le stencil test pour n'illuminer que la partie concernée, ce qui fonctionne tout autant, en diminuant le vertex count. * Ensuite, j'ai viré le maximum d'appels OpenGL de l'intérieur de la boucle des lumières (tous les états qui ne changent pas dans la boucle elle-même). * Puis j'ai modifié la fonction attenuate() dans le shader lpass.frag afin de clamper le résultat entre 0 et 1. Ca rend les artéfacts beaucoup moins perceptibles (ils apparaissent foncés au lieu d'être dans la couleur complémentaire de la surface, donc la couleur qui contraste le plus).

Ces artéfacts ne sont visibles que lors d'un mouvement de la caméra (en rotation comme en translation). *Tout se passe comme si le stencil de la frame précédente était appliqué sur la frame courante.* C'est très visible quand je profile le programme avec valgrind, ce qui a pour conséquence de le faire lagger à mort.

FIXED Il suffisait de blit depth dans LightingRenderer::render(), afin de ne pas baser les calculs de stencil sur le depth buffer de la frame précédente…

```
    // ...
    vertex_array_.bind();
    source.bind_as_source();
    target.bind_as_target();
    source.blit_depth(target);
    glClear(GL_COLOR_BUFFER_BIT);
    // ...
```

Le rendu de light volumes est donc fonctionnel, cependant il doit être optimisé à blinde. La texture bright devrait être générée en une seule passe, par exemple

[Profiling]

J'utilise *valgrind/callgrind* et un utilitaire sympa nommé *gprof2dot* pour transformer le callgrind.out.x en dot file via graphviz, après quoi je peux le convertir en image vectorielle svg grâce à *dot*.

Installer gprof2dot :

```
>> sudo -H pip install gprof2dot
```

Lancer le jeu en mode profiling (depui le dossier build) :

```
>> valgrind --tool=callgrind ../bin/wcore
```

Produire le svg en une passe :

```
>> gprof2dot --format=callgrind ./callgrind.out.[xxxxx] | dot -Tsvg -o
   callgrind.svg
```

Remplacer le [xxxxx] par les vrais chiffres en fin de fichier (je suppose que c'est le PID du process qui est utilisé). Appeler gprof2dot avec l'option -s (strip) pour virer les arguments template des noms de fonction.

Pour avoir une granularité suffisante lors du profiling tout en restant proche du build release, on compile wcore avec les options suivantes :

```
>> -O2 -g -fno-omit-frame-pointer -fno-inline-functions -fno-optimize-sibling-calls
```

Donc on active le build type “RelWithDebInfo” que j'ai configuré à ces fins, dans le CMakeLists.txt avec :

```
set(CMAKE_BUILD_TYPE RelWithDebInfo)
```

En pratique j'utilise la ligne suivante pour obtenir le SVG :

```
>> gprof2dot --format=callgrind -s --skew=0.1 ./callgrind.out.29964
| dot -Tsvg -o callgrind.svg
```

Le paramètre –skew permet d'augmenter le contraste de couleurs pour les faibles intensités quand il est <1.

Heureusement, je n'apprends rien d'énorme. On passe beaucoup de temps à calculer des produits de matrices (no shit Sherlock), et la fonction d'update d'AABBs est méga pas optimisée (mais c'est grosso merdo la seule avec un profil un peu hot). Puis on passe beaucoup de temps à dl_runtime_resolve (probablement GLEW/OpenGL).

[28-07-18]

J'ai corrigé la génération de sphères dans la mesh factory (ce qui veut dire que ma géométrie proxy était fausse mais donnait quand même de bons résultats). J'ai simplement créé une fonction pour générer un mesh de sphère solide affichable (*Vertex3P3N3T2U*, ne gère pas encore le texturing) en me basant sur la fonction de génération de sphère 3P, puis je l'ai corrigée jusqu'à ce que l'affichage d'une sphère sur la scène soit correcte, et j'ai reporté les modifications dans la fonction de génération de sphères 3P.

J'ai mesuré les temps moyens de rendu (avec écart type) pour essayer de comprendre pourquoi ma light pass stencil-optimisée rame comme une connasse.

Méthodo : * 5 lumières "dynamiques" * Caméra fixe à un endroit où 2 lumières se croisent à répétition * Exit après 1200 frames (define **PROFILING_STOP_AFTER_X_SAMPLES**) * Moyennes et écart-types calculés sur les derniers 1000 points * Testé sur ma GeForce GTX 650 (rev a1) (assez vieux matos)

Résultats : * Si on compare l'ancienne light pass (une seule passe, toutes les lumières passées dans le shader et test dynamique de distance) avec la nouvelle, la nouvelle prend environ 4.4ms de plus. * Le trait d'optimisation **OPTIM_LIGHT_VOLUMES_STENCIL_INVERT** ne change absolument rien. * Rendre des sphères ou bien le quad écran dans la light pass ne change pas le temps de rendu. * Désactiver la stencil pass complètement *ne change rien* au temps de rendu ! -> Tout se passe comme si on n'avait pas d'*early stencil test*. -> Pourtant le stencil mask est à 0 pendant la light pass. * Forcer le early fragment test avec l'instruction ci-dessous (il faut passer les shaders lpass_exp en version 420 core), ne change rien du tout.

```
    layout (early_fragment_tests) in;
```

- Faire une bright pass séparée après la light pass en utilisant un shader dédié est en fait plus lent d'environ 1ms que de produire la bright texture par blending en multipass multitarget.
- Mettre le bright threshold à 0 implique que tout l'écran sera flouté par le bloom renderer, et augmente d'environ 1ms le temps de rendu.
- Un test un peu plus à l'arrache a cependant montré que le renderer expérimental supporte sans trop de problème 64 lumières dynamiques vues de loin, là où le renderer de base décroche complètement. Ce test ne dit rien sur ce qui se passe dans le champ d'action des lumières, où c'est plutôt le culling qui fait le gros de l'optimisation. Et le culling, je ne veux pas l'implémenter dans le renderer mono-passe, trop d'emmerdes.

Je pense de toute façon qu'il va falloir programmer des *lightmaps*, pour que les lumières fixes puissent être baked.

J'ai réduit la qualité de la texture bloom qui fait maintenant la moitié de la taille de l'écran, les textures intermédiaires de la passe horizontale sont également réduites de moitié et je n'utilise plus que 3 passes blur. J'arrive à une baseline de 8.3ms pour le rendu expérimental contre 4.9ms pour le rendu mono-passe. J'ai fait un peu de place, ni plus ni moins.

Je gagne environ 300µs en virant **complètement** le contenu de null.frag.

Le calcul du fog est maintenant full post-processing. Je vais lire dans la texture depth stencil du LBuffer pour obtenir la profondeur. La profondeur obtenue est non-linéaire, du fait de la perspective (ce qui permet par ailleurs d'avoir une meilleure précision en profondeur pour les fragments proches). Elle doit donc être convertie dans les coordonnées NDC dans un premier temps, puis re-linéarisée en appliquant la transformation inverse. La donnée de profondeur obtenue est alors utilisée dans le calcul du fog, comme l'était la distance caméra-fragment précédemment :

```
1     float depthNDC    = 2.0*texture(depthStencilTex, texCoord).r - 1.0;
      float linearDepth = (2.0 * NEAR * FAR) / (FAR + NEAR - depthNDC * (FAR -
         NEAR));
```

```
    float fogFactor   = 1.0/exp(pow((linearDepth * rd.f_fogDensity),3));
    fogFactor         = clamp(fogFactor, 0.0, 1.0);
    mapped            = mix(rd.v3_fogColor, mapped, fogFactor);
```

[30-07-18]

Aujourd'hui j'ai implémenté le PBR rendering ! Enfin !

[PBR]

J'ai conservé mes choix pour la plupart des modèles : * BRDF microfacettes Cook-Torrance * Distrubution normalisée GGX Trowbridge-Reitz * Fonction géométrique GGX Schlick-Smith) * Fresnel par approximation de Schlick modifiée Gaussienne Sphérique

Tout est codé dans lpass_exp.frag en me basant sur les articles : https://learnopengl.com/PBR/Lighting https://de45xmedrsdbp.cloudfront.net/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf

Côté moteur, les modifications ont été mineures. Je n'avais qu'à définir de nouvelles textures "mandatory" dans texture.cpp, basiquement, et ajouter un canal au G-Buffer temporairement. La texture nommée "pbrTex" du G-Buffer contient les composantes métallique, rugosité et occlusion ambiente. Toutes ces textures seront packées plus sérieusement plus tard. [x] done

Comme d'habitude, tout s'est super bien passé à part un truc qui m'a fait **chier** 3h. D'étranges bandes noires / cercles noirs apparaissaient sur les faces des cristaux, lorsque je faisais le tour de ceux-ci alors qu'ils étaient éclairés par derrière. J'ai assez rapidement identifié que la fonction Fresnel produisait cet artéfact, mais je ne comprenais pas pourquoi. Puis je me suis souvenu que j'avais modifié le type de la texture normale du G-Buffer de RGBA16_SNORM à RGBA16, quand j'en ai refait le packetage lors de mes dernières optimisations. Ce faisant *mes normales perdaient gravement en précision*, et je m'en suis rendu compte en observant une discontinuité dans l'évolution des normales d'une face d'un cube en rotation grâce au debug overlay. J'ai d'abord pensé que la compression des normales en était responsable, mais non, c'était bien le type sous-jacent.

Une fois le problème réglé, j'ai eu la plus belle expérience visuelle depuis Fallout 4. Peut-être pas non plus, mais j'étais fier. Les réflexions spéculaires ont gagné en intensité et en précision, l'effet bloom par dessus donne un très bel effet sur les pavages.

Par contre c'est beaucoup plus gourmand que Blinn-Phong.

Composition du G-Buffer :

```
|___R___|___G___|___B___|___A___|
|       Position        | Rough | -> GL_RGBA16F
|___x_______y_______z___|_______|
|     Normal    | Metal |  AO   | -> GL_RGBA16_SNORM
|___x_______y___|_______|_______|
|        Albedo         |Overlay| -> GL_RGBA16
|___r_______g_______b___|_______|
```

[02-08-18]

Soft shadows. Bim.

[Soft Shadows]

Encore une fois, tout a fonctionné du premier coup, sauf qu'un bête oubli m'a fait galérer longtemps. La nouvelle classe *ShadowMapRenderer* est un *SubRenderer*. Les sub-renderers sont des renderers pour lesquels les membres VAO et Buffer Unit sont en fait des références vers les VAO et Buffer unit d'un renderer parent. *ShadowMapRenderer* est donc initialisé avec comme parent *GeometryRenderer*, ainsi il peut accéder à la géométrie de la scène. *ShadowMapRenderer* intervient dans la boucle d'illumination, et est passé comme argument à la fonction render() de *LightingRenderer*. Pour l'instant les ombres ne sont implémentées que pour la lumière directionnelle, mais l'idée est de réutiliser une même texture shadow map (que j'ai abstraite dans une classe *ShadowBuffer*). Par conséquent, je ne peux pas les calculer à l'avance dans la passe géométrique et je suis contraint à refaire des sous-passes géométriques lors de la lighting pass. Pas hyper élégant mais on fera avec.

Dans la fonction render() de *LightingRenderer* on a la ligne suivante juste avant la directional light pass :

```
1     math::mat4 lightMatrix(biasMatrix*smr.render(0));
```

smr le *ShadowMapRenderer* est appelé pour rendre la scène du point de vue de la lumière d'indice 0 (la lumière directionnelle). Ceci est effectué au moyen d'une seconde caméra de la scène que je surnomme *light_camera*, et pour laquelle j'ai codé des accesseurs dans la scène. Cette light cam est positionnée dans l'axe de la lumière directionnelle et regarde vers la position projetée verticalement de la caméra principale (pour l'instant elle regarde un endroit fixe de la scène mais osef). Elle est initialisée avec une projection orthographique grâce à un appel :

```
1     scene_.setup_light_camera(lightIndex);
```

Cette nouvelle fonction de la *Scene* appèle une fonction virtuelle de la lumière désignée par lightIndex pour initialiser la light cam. Selon le type de lumière on veut des projections et des positionnements différents:

```
1     void Scene::setup_light_camera(uint32_t index)
      {
3         const math::vec3& campos = camera_->get_position();
          get_light(index)->setup_camera(*light_camera_,
              vec3(campos.x(),0.0f,campos.z()));
5     }
```

Et en l'occurrence, pour une lumière directionnelle il se passe ça :

```
1 void DirectionalLight::setup_camera(Camera& camera, const math::vec3& posLookAt)
     const
      {
3         camera.set_position(position_+posLookAt);
          camera.look_at(posLookAt);
5         camera.set_orthographic(1024, 1024, 25);
      }
```

Le 25 dans set_orthographic correspond à un niveau de zoom, qui va in fine diviser d'autant les composantes diagonales en x et y.

Une fois la light cam initialisée, ses matrices de vue et projection sont récupérées et la scène est dessinée de son point de vue, mais uniquement dans le depth buffer, qui est une texture nommée "shadowmap". Le null shader (null technique) est utilisé ici. Les modèles de la scène sont dessinés soit en cull_front soit en cull_back, selon comment ils ont été initialisés (en effet, on veut cull_back sur les modèles convexes et cull_front sur les autres, comme le terrain typiquement, afin d'éviter respectivement le *peter-panning* et la *shadow acnee*). La matrice vue-projection de la light cam est alors retournée à l'appelant par ShadowMapRenderer::render().

Ainsi, dans LightingRenderer::render(), on obtient la matrice de vue projection pertinente pour la reprojection de la shadow map, juste après que celle-ci a été générée. A noter que cette matrice doit être pré-multipliée par une *bias matrix* de manière à remapper les coordonnées du système NDC (view space $[-1, 1]^3$) vers l'espace UV (texture space $[0, 1]^3$). Cela permet d'économiser l'opération :

```
shadowMapCoords = 0.5 * shadowMapCoords + 0.5;
```

dans le shader pour chaque fragment. La shadow map doit être bind en tant que texture et le sampler shadowTex associé à la texture active d'indice 3 :

```
    glActiveTexture(GL_TEXTURE3);
    glBindTexture(GL_TEXTURE_2D, (*pshadow)[0]);
    lighting_pass_shader_.send_uniform_int("shadowTex", 3);
```

(En passant, c'est ça que j'avais oublié de faire, et je note également que c'est bien en **int** et non en uint qu'il faut envoyer les uniforms samplers… fuck.) Trois uniforms supplémentaires sont envoyés au lighting_pass_shader_ :

```
    lighting_pass_shader_.send_uniform_mat4("m4_LightSpace", lightMatrix);
    lighting_pass_shader_.send_uniform_vec2("rd.v2_shadowTexelSize",
       ShadowMapRenderer::SHADOW_TEXEL_SIZE);
    lighting_pass_shader_.send_uniform_float("rd.f_shadowBias", 0.4f/1024.0f);
```

La fameuse matrice de vue projection nommée ici lightMatrix (prémultipliée par la bias matrix), la taille en (x,y) d'un texel sur la shadow map (afin de calculer correctement les offsets pour le multisampling PCF), et un paramètre de biais pour décaler uniformément en z la valeur de la shadow map, et pallier certaines pathologies typiques du shadow mapping (peter-panning et shadow acnee). La valeur de 0.4 a été trouvée expérimentalement comme bon compromis.

Dans le shader lpass_exp.frag on n'a qu'à transformer les coordonnées monde de chaque fragment (depuis le G-Buffer) dans le repère de la lumière au moyen de la lightMatrix. Puis tester si le fragment est à une profondeur supérieure à celle enregistrée dans la shadowmap (signifiant que le fragment est dans l'ombre). Techniquement on fait ça pour tout un voisinage autour de la position à tester, et on retourne une valeur entre 0 et 1, combinaison linéaire de tous les échantillons.

[05-08-18]

[TODO][x] Faire 2 shaders séparés pour les lumières directionnelle et ponctuelles.

```
[x] Pour éviter d'avoir à copier-coller toutes les fonctions de lighting,
   améliorer la classe _Shader_ pour qu'elle puisse gérer des "includes".
-> Voir shader variants.
```

[x] Optimiser les ombres directionnelles [x] Contraindre la lightcam à observer la projection du frustum de la cam principale sur le sol. [o] Near et Far de la lightcam en tight fit également depuis des données de la scène. [ ] Implémenter des depth cubemaps. [ ] Avec un *BufferModule* ? Ou classe séparée ? [ ] Pour chaque lumière ponctuelle (configurée pour projeter une ombre) rendre une depth cubemap en une seule passe avec le *geometry shader trick*. En effet, un geometry shader peut (ssi une cubemap est attachée au FBO courant) sélectionner une des faces du cube sur laquelle envoyer des primitives au moyen de **gl_Layer**. [ ] 6 light space matrices sont générées (une pour chaque face, alignées avec le repère monde), lesquelles sont composées d'une unique projection perspective et d'une matrice de vue (une pour chaque face). Le FoV **doit être de 90°**. [ ] Le vertex shader transforme les positions vers l'espace monde seulement (multiplication par la matrice modèle). [ ] Le geometry shader a accès aux 6 light space matrices, et pour chaque face de la cubemap, émet 3 vertices transformés avec la light matrix correspondante. Donc basiquement on réplique la géométrie 6 fois. [ ] Cette fois un fragment shader non vide va calculer des profondeurs linéaires (distance source-fragment normalisée entre [0,1] (division par Far)) et les écrire dans *gl_FragDepth*. [ ] Puis utiliser cette shadow map dans la passe lighting pour implémenter les ombres omnidirectionnelles.

https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

[17-08-18]

J'ai bossé les 3 derniers jours après une pause Far Cry 4 (nom de Dieu, ce jeu !). Essentiellement, j'ai construit un parser XML pour la scène, corrigé quelques inélégances du code, et préparé le terrain pour un système de chunks.

[XML] Format de map

La classe *SceneLoader* utilise la lib externe (.hpp only) RapidXML pour parser (à toute vitesse si l'on en croit les promesses du site) un fichier XML contenant la description de la scène. Le fichier est dans un premier temps copié dans un buffer (qui devra survivre aussi longtemps que le DOM, car RapidXML est un parser *in-situ*), puis le DOM est généré. Le format de ma scène est une solution "roll my own shitty one", on verra pour la putain d'usine à gaz Collada plus tard. Le noeud principal est nommé *Scene*, et contient essentiellement des noeuds *Chunk* et un autre noeud *Camera* (et dans l'idée plus tard, d'autres objets "globaux" de la scène).

```
    <Camera>
        <Position>(0.0,1.2,-2.5)</Position>
        <Orientation>(229.0,27.0)</Orientation>
    </Camera>
```

Pour l'instant les interpolateurs Bézier et le code d'upload est toujours codé en dur, j'ai peur qu'il ne me faille un système de motion integration plus complexe pour changer celà. On pourrait faire des objets mouvants de la scène des entités à part entière et leur ajouter un composant *ComponentMotion* ou je ne sais quoi. Mais en l'état la *Scene* ne sait pas ce qu'est une entité (je n'utilise pas encore mes classes ECS, bouuuuuh).

[Chunk]

Chaque *Chunk* possède un attribut "id", lequel est sauvegardé par *SceneLoader* dans une map de pointeurs vers les *Chunks* du DOM. Plusieurs noeuds enfants peuvent être rencontrés dans un noeud *Chunk*, et tous sont optionnels :

```
<?xml version="1.0" encoding="utf-8"?>
<Scene>
    <Chunk id="0">
```

[Terrain]

Contient des noeuds *TerrainPatch* avec les attributs "width", "length" et "height". width et length sont les tailles entières de la heightmap utilisée pour générer le patch terrain, et height la hauteur (float) initiale du terrain. Un noeud *TerrainPatch* contient les noeuds suivants : * *HeightModifier* qui pour l'instant ne peut contenir que des noeuds *Randomizer* qui permettent de randomiser la hauteur d'une partie souhaitée du terrain

```
    <Terrain>
        <TerrainPatch width="32" length="32" height="0">
            <HeightModifier>
                <Randomizer seed="1" xmin="0" xmax="32" ymin="0" ymax="16"
                    variance="0.07"></Randomizer>
                <Randomizer seed="2" xmin="0" xmax="32" ymin="17" ymax="32"
                    variance="0.25"></Randomizer>
            </HeightModifier>
```

- *Transform* dont l'enfant *Position* définit la position du coin (0,0) en world space (ici pour avoir l'origine world au milieu du patch)

```
            <Transform>
                <Position>(-16,0,-16)</Position>
            </Transform>
```

- *Material* qui rassemble les propriétés d'un *Material*. On peut spécifier un asset ou bien des propriétés homogènes.

```
            <Material>
                <Asset>pavedFloor</Asset>
                <NormalMap>true</NormalMap>
                <ParallaxHeightScale>0.15</ParallaxHeightScale>
            </Material>
```

- *AABB* permet uniquement de décaler l'AABB associé au terrain (ce qui est nécessaire uniquement pour les terrains pour une raison qui m'échappe encore).

```
        <AABB>
            <Offset>(7.75,-0.125,7.75)</Offset>
        </AABB>
```

- *Shadow* qui permet pour l'instant uniquement de sélectionner les faces à cull lors du rendu de l'ombre (0=cull désactivé, 1=front, 2=back)

```
        <Shadow>
            <CullFace>1</CullFace>
        </Shadow>
    </TerrainPatch>
</Terrain>
```

[Models]

Ne contient que des noeuds *Model* :

```
<Models>
    <!-- Big Erwin cube -->
    <Model id="0">
        <Mesh>cube</Mesh>
        <Material>
            <Asset>cube</Asset>
            <Overlay>true</Overlay>
        </Material>
        <Transform>
            <Position>(0.0,0.0,-2.0)</Position>
            <Scale>1.0</Scale>
        </Transform>
    </Model>
```

En général, un modèle regroupe essentiellement 3 composantes : * Un ou plusieurs meshes * Un ou plusieurs materials * Une ou plusieurs transformations Comme mes modèles ne comportent qu'un couple mesh/material et une unique transfo pour l'instant je n'ai pas fait compliqué, rien n'empêche de complexifier la structure par la suite pour refléter une hiérarchie dans le modèle.

Autre exemple avec un modèle dont le material définit des valeurs homogènes plutôt que des maps :

```
        <!-- Axis aligned colored cubes -->
        <Model id="3">
            <Mesh>cube</Mesh>
            <Material>
                <Color>(1,0,0)</Color>
                <Roughness>0.3</Roughness>
            </Material>
            <Transform>
                <Position>(1,0,0)</Position>
                <Scale>0.3</Scale>
            </Transform>
        </Model>
```

[ModelBatches]

Me permet de générer plusieurs modèles à la volée (lesquels peuvent être procéduraux). Le code est assez parlant :

```
<ModelBatches>
    <ModelBatch instances="25" seed="48">
        <Mesh>crystal</Mesh>
        <Material>
            <Color space="hsl" variance="(0.15,0.1,0.0)">(0.45,0.9,0.5)</Color>
            <Roughness>0.3</Roughness>
        </Material>
        <Transform>
            <Position variance="(7.0,0.0,6.5)">(8.0,-0.7,8.0)</Position>
            <Angle variance="(15.0,90.0,5.0)">(15.0,0.0,0.0)</Angle>
            <Scale variance="0.4">0.8</Scale>
        </Transform>
        <Shadow>
            <CullFace>2</CullFace>
        </Shadow>
    </ModelBatch>
</ModelBatches>
```

Ce que j'ai (très mal) nommé "variance" V correspond en réalité au supremum d'une distribution uniforme centrée sur 0 ($X_{instance} = X \pm V$). Pour chaque instance du batch, une valeur aléatoire est calculée à partir de ce paramètre, et est ajoutée à la valeur centrale spécifiée comme valeur du noeud. En pratique ma distribution est uniforme dans $[-1, 1]$ et est multipliée par le paramètre V pour l'amener dans $[-V, V]$.

Ici, comme partout où un attribut **seed** est spécifié, on aura côté C++ l'initialisation d'un Mersenne Twister std::mt19937 avec la seed extraite. Toutes les valeurs pseudo-aléatoires dans le scope seront alors produites par différentes distributions via ce même générateur, ainsi on observera toujours le même résultat pour la même seed.

```
uint32_t instances, seed;
xml::parse_attribute(batch, "instances", instances);
xml::parse_attribute(batch, "seed", seed);
std::mt19937 rng;
rng.seed(seed);
std::uniform_int_distribution<uint32_t>
   mesh_seed(0,std::numeric_limits<uint32_t>::max());
std::uniform_real_distribution<float> var_distrib(-1.0f,1.0f);
// ...
pmesh = factory::make_crystal(mesh_seed(rng));
// ...
vec3 color, color_var;
xml::parse_node(mat_node, "Color", color);
xml::parse_attribute(mat_node->first_node("Color"), "variance", color_var);
// ...
vec3 inst_color = color + vec3(color_var.x() * var_distrib(rng),
                               color_var.y() * var_distrib(rng),
                               color_var.z() * var_distrib(rng));
// ...
std::string color_space;
xml::parse_attribute(mat_node->first_node("Color"), "space", color_space);
if(!color_space.compare("hsl"))
    inst_color = color::hsl2rgb(inst_color);
```

Au passage on notera que je peux spécifier mes couleurs dans l'espace colorimétrique HSL (en plus de RGB). C'est un atout évident quand on veut pouvoir spécifier des couleurs pseudo-aléatoires perceptivement différentes (une couleur RGB aléatoire aura tendance à être pastel dégueu, tandis qu'avec HSL on peut spécifier un intervalle de teinte…) ! Voir la section [Color] HSL.

[**Lights**]

Là encore, c'est très direct :

```
    <Lights>
        <!-- Sun -->
        <Light id="6" type="directional">
            <Position>(0.0,1.0,1.0)</Position>
            <Color>(0.25,0.35,0.75)</Color>
            <Brightness>2.0</Brightness>
        </Light>

        <!-- Point light -->
        <Light id="7" type="point">
            <Position>(0,0,0)</Position>
            <Color>(0.05, 0.92, 0.87)</Color>
            <Radius>5.0</Radius>
            <Brightness>10.0</Brightness>
        </Light>
    </Lights>
```

[Parsing]

Dans le fichier xml_utils.hpp j'ai écrit quelques fonctions templates qui simplifient le parsing des noeuds. Essentiellement, les fonctions parse_attribute() récupèrent la valeur d'un attribut donné d'un noeud donné pour la mettre dans une variable donnée, et les fonctions parse_node() récupèrent la valeur d'un noeud donné d'un parent donné pour la mettre dans une variable donnée. Le template argument deduction fonctionne pleinement et on peut écrire ce genre de code :

```
    std::string asset;
    bool use_normal_map = false;
    float parallax_height_scale = 0.1f;
    success = true;
    success &= xml::parse_node(mat_node, "Asset", asset);
    success &= xml::parse_node(mat_node, "NormalMap", use_normal_map);
    success &= xml::parse_node(mat_node, "ParallaxHeightScale",
       parallax_height_scale);
    if(!success) suck_a_dick_and_die();
```

Noter que les spécialisations complètes des templates de xml_utils.hpp sont déclarées dans ce fichier mais *définies* dans xml_utils.cpp (afin d'éviter un ennuyeux problème de définitions multiples).

[Color] HSL

Les fichiers colors.h/cpp définissent dans le namespace "color" deux fonctions pour la conversion des couleurs de HSL vers RGB et inversement. J'ai codé ça moi-même après une lecture des maths sous-jacentes sur Wikipedia.

La transformation est en fait assez simple à intuiter. L'espace RGB est un cube (borné) dont le vertex noir est à l'origine (0,0,0). Imaginons que l'on effectue une rotation du cube autour du point d'origine qui amène le cube au dessus du plan (xy) et le vertex blanc (1,1,1) opposé sur l'axe z (équivalent à un choix de *coordonnées Cartésiennes auxilliaires*). La projection du cube sur le plan (xy) est un hexagone avec les vertices rouge, jaune, vert, cyan, bleu, et magenta. Cet hexagone est alors remappé sur un cercle paramétré par l'angle au centre theta, angle qui correspond alors à la teinte (Hue).

Pour la luminosité (Lightness), une définition doit être choisie parmi plusieurs possibilités. J'utilise la norme (perceptivement pertinente) Luma Y_709 pour les primaires sRGB (par cohérence avec mes shaders) qui est le produit scalaire de la couleur RGB avec (0.2126, 0.7152, 0.0722).

La saturation est calculée depuis la luminosité et une valeur intermédiaire (chroma) qui est simplement la distance à l'origine.

```
static const float SQ3_2 = 0.866025404f;
static const math::vec3 W(0.2126, 0.7152, 0.0722);

math::vec3 rgb2hsl(const math::vec3& rgb_color)
{
    // Auxillary Cartesian chromaticity coordinates
    float a = 0.5f*(2.0f*rgb_color.r()-rgb_color.g()-rgb_color.b());
    float b = SQ3_2*(rgb_color.g()-rgb_color.b());
    // Hue
    float h = atan2(b, a);
    // Chroma
    float c = sqrt(a*a+b*b);
    // We choose Luma Y_709 (for sRGB primaries) as a definition for Lightness
    float l = rgb_color.dot(W);
    // Saturation
    float s = (l==1.0f) ? 0.0f : c/(1.0f-fabs(2*l-1));
    // Just convert hue from radians in [-pi/2,pi/2] to [0,1]
    return math::vec3(h/M_PI+0.5f,s,l);
}
```

Noter que la teinte qui mathématiquement est un angle en radians dans $[-\pi/2, \pi/2]$ est remappée vers $[0, 1]$ pour plus de simplicité.

Pour faire le chemin dans le sens inverse avec hsl2rgb() on notera d'abord que toute couleur RGB peut s'écrire comme la somme d'un scalaire m (offset de luminosité) et d'un 3-vecteur dont un élément est 0 et les deux autres non nuls correspondent à la chroma C et au deuxième canal le plus intense moins la luminosité (noté X) :

```
(R,G,B) = m + (C,X,0)
       ou m + (X,C,0)
       ou m + (0,C,X)
       ou ...
```

Je calcule donc la chroma C depuis la représentation HSL et un intermédiaire X, la deuxième couleur RGB la plus forte, puis je détermine dans quelle portion de l'hexagone projeté je suis (grâce à la teinte qui n'est autre que l'angle au centre), pour décider de la permutation des C et X dans la représentation RGB, avant d'ajouter l'offset de luminosité calculé depuis la luminosité et la chroma.

```
math::vec3 hsl2rgb(const math::vec3& hsl_color)
{
    // Chroma
    float c = (1.0f-fabs(2.0f*hsl_color.z()-1.0f))*hsl_color.y();
    // Intermediate
    float Hp = hsl_color.x()*6.0f; // Map to degrees -> *360°, divide by 60° -> *6
    float x = c*(1.0f-fabs(fmod(Hp,2) - 1.0f));
    uint8_t hn = uint8_t(floor(Hp));

    // Lightness offset
    float m = hsl_color.z()-0.5f*c;

    // Project to RGB cube
    switch(hn)
    {
        case 0:
            return math::vec3(c+m,x+m,m);
        case 1:
            return math::vec3(x+m,c+m,m);
        case 2:
            return math::vec3(m,c+m,x+m);
```

```
        case 3:
            return math::vec3(m,x+m,c+m);
        case 4:
            return math::vec3(x+m,m,c+m);
        case 5:
            return math::vec3(c+m,m,x+m);
        default:
            return math::vec3(m,m,m);
    }
}
```

Je ne sais pas si je vais garder ça comme ça, ça fonctionne nickel mais c'est cheum…

Refactor

Plusieurs modifications ont été apportées à la codebase pour simplifier des dépendances inutiles.

[Singletonization]

GBuffer et LBuffer étaient des candidats idéaux à la globalité. Seulement dans un souci de RAII, ces deux objets n'ont pas de constructeur par défaut (ainsi que toutes leurs classes parentes). Alors j'ai écrit un deuxième template de singletons, la classe *SingletonNDI* (NDI pour Non-Default Initialized) qui prévoit une fonction template variadique qui forward les paramètres d'initialisation au constructeur de la sous-classe singleton :

```
    template <typename... Args>
    static void Init(Args&&... args)
    {
        if(instance_ == nullptr)
            instance_ = new T(std::forward<Args>(args)...);
    }
```

Pour friender une telle fonction il faut déclarer ceci dans la sous-classe :

```
    template <class T, typename ...Args>
    friend void SingletonNDI<T>::Init(Args&&... args);
```

Ceci est du code C++ *parfaitement valide* mais malheureusement non supporté par Clang qui émet un warning -W-unsupported-friend et avertit que le contrôle d'accès est désactivé pour cette fonction. Comprendre que son accès sera public, ce qui en dernière instance ne me pose pas de problème ; j'ai donc **désactivé le warning** avec -Wno-unsupported-friend.

voir : https://stackoverflow.com/questions/37115503/friend-of-function-in-dependent-scope

Ainsi, les objets *GBuffer* et *LBuffer* sont maintenant globalement uniques et il n'est plus nécessaire de les passer en paramètre des fonctions render() de chaque renderer. Toutes les fonctions render() ont même signature et peuvent maintenant redevenir virtuelles (à l'exception de celle de *LightingRenderer* pour l'instant, mais on y reviendra).

[BufferUnit] Leur place est-elle bien dans un renderer ?

Contrairement à disons il y a 3-4 ans, je ne suis plus un nazi du paradigme de programmation. Je pense qu'il est souhaitable qu'il y ait une séparation code/données là où ça fait sens, et pas de séparation là où ça complique les choses.

Donc mes renderers vont bel et bien conserver leurs couples *VertexArray*/*BufferUnit* par défaut (en l'état de ma réflexion), car il est vrai que certaines géométries sont immuables sur toute leur durée de vie (le quad écran, les géométries proxy du *LightingRenderer*…). En revanche, la géométrie de la scène doit être possédée par la *Scene* et pas par le *GeometryRenderer*, nom de Dieu !

En réfléchissant à un système de chunks, il m'a semblé évident que chaque chunk doit posséder un *BufferUnit* propre. En effet, il n'est pas sérieux de changer TOUT le contenu d'un unique vertex buffer dès qu'un nouveau chunk est chargé. Donc j'ai ajouté deux *BufferUnits* dans la *Scene* (un pour les modèles opaques et un pour les modèles avec blending) ainsi qu'un *VertexArray*. Deux renderers sont affectés par cette modification : *GeometryRenderer* et *ForwardRenderer*, lesquels utilisent maintenant leur référence à la *Scene* pour manipuler les vertex array et buffer units pertinents.

[Simplifications]

- Donc comme *GeometryRenderer* n'héberge plus la géométrie de la scène (son *BufferUnit* est vide), le *ShadowMapRenderer* n'a plus accès à la géométrie de la scène et ne dessine plus d'ombre. Donc on vire son héritage à *SubRenderer* qui n'a plus aucun sens, on vire la classe *SubRenderer* qui elle-même n'a plus aucune raison d'être, et on utilise l'interface de la *Scene* pour pouvoir dessiner la shadow map. *ShadowMapRenderer* n'est en l'état même pas un *Renderer*, c'est pour l'instant une classe stand alone. Il n'est d'ailleurs plus question de la passer en paramètre à la fonction render() de *LightingRenderer*. Elle lui est passée lors de la construction, à l'instar de *TextRenderer* qui est passée au constructeur de *DebugOverlayRenderer*.
- La *Scene* enregistre maintenant la taille de l'écran. Beaucoup de renderers ont en effet besoin de la taille de l'écran pour leur initialisation, mais TOUS ont besoin de la *Scene*. Il me suffit d'accéder à la taille de l'écran à travers la *Scene* et je simplifie nombre de constructeurs chez les renderers.
- La lumière directionnelle est unique dans la *Scene* donc plus d'hypocrisie en la fourrant dans le vector de *Light* à la position 0. C'est une lumière spéciale donc on y accède spécialement (avec add_directional_light() et get_directional_light() pour être spécifique).

[Chunk System] Préparation

A terme, la *Scene* possédera une collection de *Chunk* et une série de fonctions pour la traverser, en abstrayant les chunks. Par exemple, on aura toujours une fonction Scene::traverse_models() qui elle-même traversera les modèles de chaque chunk itérativement. Dans le cas des visiteurs traversant une permutation des objets de la scène (comme traverse_models_sorted_fb()), j'imagine stocker une liste de permutation par chunk et trier les chunks par proximité également.

Les membres suivants de la *Scene* deviendront ceux de *Chunk* :

```
    std::vector<pModel> models_;
    std::vector<pModel> models_blend_;
    std::vector<pLight> lights_;
    std::vector<uint32_t> models_order_;
    std::vector<uint32_t> blend_models_order_;
    BufferUnit<Vertex3P3N3T2U>  buffer_unit_;
    BufferUnit<Vertex3P3N3T2U>  blend_buffer_unit_;
```

La *Scene* conservera les membres suivants :

```
    VertexArray<Vertex3P3N3T2U> vertex_array_;
    pLight directional_light_;
    pCamera camera_;
    pCamera light_camera_;
    uint32_t screen_width_;
    uint32_t screen_height_;
```

Noter qu'on est très content d'avoir séparé les VAO des VBO/IBO. Le vertex array de la *Scene* est compatible avec les *BufferUnit* de chaque *Chunk* donc on économise des binds.

Tous les modèles de la scène ne seront pas associés à un chunk (le vaisseau du joueur, certains ennemis spéciaux dont les boss…). Ces derniers pourraient être stockés dans l'objet *Scene* lui-même, donc pas impossible que je me retrouve avec un *BufferUnit* ou deux dans la *Scene* à terme, quand même. J'envisage d'utiliser le système de chunks

pour les objets statiques ou proche de l'être, en tout cas, tous les objets qui peuvent être détruits en même temps que le chunk auquel ils appartiennent quand celui-ci sort du champ de vision, sans pour autant disparaitre à l'écran. Tant que ce n'est pas trop fréquent, on peut imaginer que certains objets puissent échapper au système de chunk et devenir persistants (jusqu'à leur destruction physique par le joueur ou leur sortie de l'écran par exemple), ceci nécessitant sans doute un re-submit. Mais on est malin quand on n'est pas défoncé, donc on va gérer ça.

[31-08-18] Des pommes, des bananes, des figues

Tant de retard dans ma doc…

[Chunk]

Echec de la tentative d'implémentation du système de chunk, j'étais trop pressé et j'ai fait n'imp', comme d'habitude. L'interface de la *Scene* projette loin dans le système. Ma meilleure option consisterait à travailler d'abord sur une nouvelle interface avant de changer la tuyauterie. Avant d'attaquer ce refactoring, je dois donc définir avec un peu plus de précision cette interface.

[Motion]

Début d'un système de motion integration. Le fichier motion.hpp regroupe un ensemble de classes pour l'interpolation des transformations de modèles.

Pour l'instant je n'anime que la position. J'ai une classe *PositionUpdater* qui possède deux membres connus à travers une interface (lien de composition) : un interpolateur de vec3 et un "opérateur d'évolution temporelle" qui spécifie comment on parcourt l'espace paramètre (de manière cyclique, alternée…).

```
    Interpolator<math::vec3>*  interpolator_;
    timeEvolution::TimeUpdater* time_updater_;
```

Voici les deux interfaces :

```
template <typename T>
class Interpolator
{
public:
    virtual T operator()(float t) = 0;
    virtual ~Interpolator(){}
};

namespace timeEvolution
{
class TimeUpdater
{
public:
    virtual ~TimeUpdater(){}
    virtual void step(float& time,
                      float dt,
                      float scale,
                      float tmin,
                      float tmax) = 0;
};
}
```

L'implémentation de l'interpolateur utilise l'héritage multiple et le polymorphisme dynamique pour sécifier le comportement de l'interpolation. J'ai pour l'instant définit une unique classe *BezierInterpolator* qui interpole un vecteur au moyen d'une courbe de Bézier :

```
class BezierInterpolator : public math::Bezier, public Interpolator<math::vec3>
```

J'envisage de faire de même avec des classes de splines.

```
Note: J'hésite à faire de _Bezier_ un _Interpolator<math::vec3>_ directement, la
    hiérarchie s'en trouverait moins encombrée.
```

Grosse parenthèse

Mon implémentation initiale différait complètement. J'avais en effet construit tout ce système avec des templates *Policy Oriented*, l'interpolation et l'évolution temporelle formaient des polices de la classe template *PositionUpdater*. C'était parfait et esthétique, jusqu'à ce que j'ai à générer ces objets dynamiquement depuis le parseur xml. Il faut spécifier chaque combinaison possible, ce qui est simplement bordélique avec du branching.

Donc j'ai envisagé de coder une *abstract factory*, laquelle pourrait enregistrer chaque combinaison et créer des objets à la demande au moyen d'un ID spécifique au type, et les présenter sous l'interface d'une classe de base. Or un problème connu (par ceux qui ont essayé) des abstract factories, est la difficulté d'y enregistrer des classes templates. C'est le *problème de la souscription*. En effet, chaque combinaison d'implémentations des polices qui paramètrent le template engendre un type C++ particulier, qu'il faut enregistrer séparément dans la factory. Donc quand on rajoute une implémentation pour une police donnée, on doit écrire autant d'instructions d'enregistrement qu'il y a de nouvelles combinaisons possibles avec les autres polices. Et l'explosion combinatoire liée au rajout d'un paramètre au template est rédhibitoire. C'est la définition pure et simple d'un code non entretenable.

Mais tous les problèmes ont leur solution : [Peleg] https://www.artima.com/cppsource/subscription_problem.html

On peut utiliser la méta-programmation pour souscrire toutes les combinaisons possibles (et même filtrer des combinaisons illégales) d'un template à une factory. L'implémentation proposée par [Peleg] montre la création d'une classe FactorySubscriber_2 capable de souscrire un template à 2 paramètres automatiquement, et c'est assez fastidieux. Il faudrait faire de même pour 3, 4, 5, ..., n paramètres. Par ailleurs l'implémentation repose sur l'utilisation des Typelists (du livre Modern C++ et son excellente lib full header associée du nom de Loki). Donc, possible, mais affreusement lourdingue.

Parfois, le bon vieux polymorphisme dynamique est de mise. A titre indicatif, voilà l'ancienne implémentation de motion.hpp :

```
#ifndef MOTION_HPP
#define MOTION_HPP

#include "bezier.h"
#include "cspline.h"

namespace timeEvolution
{
class Alternating
{
private:
    bool forward_;

public:
    Alternating(): forward_(true) {}

    void step(float& time, float dt, float scale, float tmin, float tmax)
    {
        if(forward_)
        {
```

```
            time += dt*scale;
            if(time>tmax)
            {
                time = tmax;
                forward_ = false;
            }
        }
        else
        {
            time -= dt*scale;
            if(time<tmin)
            {
                time = tmin;
                forward_ = true;
            }
        }
    }
};

class Cyclic
{
public:
    void step(float& time, float dt, float scale, float tmin, float tmax)
    {
        time += dt*scale;
        if(time>tmax)
            time = tmin;
    }
};
}

template <typename T>
class IMotionUpdater
{
protected:
    float t_; // Internal clock
public:
    IMotionUpdater():
    t_(0.0f){}

    virtual ~IMotionUpdater(){}

    inline void set_time(float tt) { t_ = tt; }
    inline float get_time() const  { return t_; }

    virtual T operator()(float dt) = 0;
};

template <typename Interpolator,
          typename TimeEvolution = timeEvolution::Alternating>
class PositionUpdater : public IMotionUpdater<math::vec3>
{
private:
    Interpolator  interpolator_;
    TimeEvolution time_updater_;
```

```
77      float scale_;
        float tmin_;
79      float tmax_;

81  public:
        PositionUpdater(const Interpolator& interpolator,
83                      float scale = 1.0f,
                        float tmin = 0.0f,
85                      float tmax = 1.0f):
        IMotionUpdater(),
87      interpolator_(interpolator),
        scale_(scale),
89      tmin_(tmin),
        tmax_(tmax)
91      {

93      }

95      inline void set_step_scale(float scale) { scale_ = scale; }
        inline float get_step_scale() const     { return scale_; }
97
        inline void set_tmin(float tmin)        { tmin_ = tmin; }
99      inline float get_tmin() const           { return tmin_; }

101     inline void set_tmax(float tmax)        { tmax_ = tmax; }
        inline float get_tmax() const           { return tmax_; }
103
        virtual math::vec3 operator()(float dt) override
105     {
            time_updater_.step(t_, dt, scale_, tmin_, tmax_);
107         return interpolator_.interpolate(t_);
        }
109 };

111 template <typename Interpolator, typename TimeEvolution>
    using ColorUpdater = PositionUpdater<Interpolator, TimeEvolution>;
113
    #endif // MOTION_HPP
```

[**XML**]

Voici la définition actuelle d'un cube texturé qui bouge dans un fichier de map :

```
        <Model>
2           <Mesh>cube</Mesh>
            <Material>
4               <Asset>cube</Asset>
                <Overlay>true</Overlay>
6           </Material>
            <Transform>
8               <Position>(16.0,0.0,14.0)</Position>
                <Scale>1.0</Scale>
10          </Transform>
            <Motion>
12              <PositionUpdater>
                    <Prop name="tSpace">alternate</Prop>
```

```
                <Prop name="stepScale">0.33</Prop>
                <Prop name="tMax">1.0</Prop>
                <Prop name="tMin">0.0</Prop>
                <BezierInterpolator>
                    <Control>(15,0.5,13.8)</Control>
                    <Control>(9,2,13.8)</Control>
                    <Control>(12,-1,14)</Control>
                    <Control>(10,1,11)</Control>
                    <Control>(8,3,8)</Control>
                </BezierInterpolator>
            </PositionUpdater>
        </Motion>
    </Model>
```

Ici on déclare que le cube parcourt une courbe de bézier spécifiée par ses points de controles, et le fait de manière alternée (suit la courbe dans un sens puis dans l'autre). L'incrément paramétrique est contrôlé à travers un paramètre d'échelle (t -> t + stepScale * dt), et les bornes de l'espace du paramètre sont aussi spécifiées dans deux autres propriétés du noeud *PositionUpdater*.

[Terrain]

Un générateur de terrain par bruit simplicial est exposé côté XML. J'ai simplement réutilisé (modifié et testé) le générateur de WEngine. La "seed", alors fixe, consistait en un tableau de permutation des 255 premiers entiers codé en dur dans noise_policy.hpp. J'ai écrit du code pour initialiser ce tableau avec une permutation pseudo-aléatoire contrôlable par un Mersenne Twister et une seed donnée :

```
    void SimplexNoise::init(std::mt19937& rng)
    {
        // Random permutation table
        for(short ii=0; ii<=255; ++ii)
            randp_[ii] = ii;
        std::shuffle(randp_.begin(), randp_.end(), rng);

        // Initialize permutation table
        for(int ii=0; ii<512; ++ii)
        {
            perm_[ii] = randp_[ii & 255];
            perm_mod_12_[ii] = (short)(perm_[ii] % 12);
        }
    }
```

La classe *NoiseGenerator2D* permet de générer du bruit d'octaves par sampling de la fonction de bruit de sa police *NoisePolicy* (implémentations existantes à ce jour : *SimplexNoise* et *CellNoise*). Le générateur d'octaves produit une somme pondérée de buits à des fréquences spatiales différentes, afin de rendre compte des multiples échelles de détail dans le façonnement du terrain. Les fichiers heightmap_generator.h/cpp implémentent des fonction pour l'initialisation d'une *HeightMap* au moyen d'un tel générateur d'octaves (dans le namespace *terrain*). Une fonction erode() permet également de simuler un type d'érosion de terrain. Dans l'idée, d'autres modifiers de ce type peuvent trouver leur place dans ces fichiers.

Je pense avoir décrit dans un papier (quelque part) le fonctionnement interne du générateur, car ça avait été un gros travail de recherche à l'époque pour mettre le truc au point (la génération de bruit de simplex est plus rapide et plus élégante que celle du bruit de Perlin, mais bien moins documentée). En attendant que je remette la main sur mes notes, je retarde les explications ici.

```
EDIT: Yep, c'est dans mon carnet noir à la date [01-08-15] :
Pour des coordonnées réelles d'entrée (x,y) on calcule le point correspondant
    (i,j) dans la base oblique de la tesselation simpliciale de l'espace ((i,j,k)
```

```
en 3D...). On détermine alors dans quel simplex on se trouve. Pour chaque
sommet du simplex on sélectionne un gradient pseudo-aléatoire par hashage de
ses coordonnées (c'est là qu'intervient la table de permutation), et on somme
les contributions de chaque sommet à l'aide de noyaux centrés à symmétrie
radiale.
```

L'avantage par rapport au bruit de Perlin c'est la complexité. Perlin est basé sur une tesselation de l'espace en hypercubes et donc sa complexité est en O(2^N) avec N la dimension de l'espace, tandis qu'un simplex est de dimension N+1, donc le bruit simplicial est en O(N+1), ce qui est radicalement plus optimal. Et on ne crache pas sur Ken Perlin pour autant, il est également l'auteur du bruit simplicial…

Le générateur *CellNoise* que je n'ai pas retouché produit du bruit cellulaire par génération de *diagrammes de Voronoï* soumis à une fonction distance donnée. Les distances de *Chebychev* et de *Manhattan* produisent de beaux artéfacts rectilignes sur le terrain.

Voici un exemple de terrain :

```
<Terrain>
    <TerrainPatch width="32" length="32" height="0" main="true">
        <Generator type="simplex" seed="2">
            <Prop name="octaves">10</Prop>
            <Prop name="frequency">0.01</Prop>
            <Prop name="persistence">0.4</Prop>
            <Prop name="loBound">0.0</Prop>
            <Prop name="hiBound">25.0</Prop>
        </Generator>
        <HeightModifier>
            <Erosion iterations="20" talus="0.4" fraction="0.5"></Erosion>
            <Randomizer seed="1" xmin="0" xmax="32" ymin="0" ymax="16"
                variance="0.07"></Randomizer>
            <Randomizer seed="2" xmin="0" xmax="32" ymin="17" ymax="32"
                variance="0.25"></Randomizer>
            <Offset y="-8.5"></Offset>
        </HeightModifier>
        <Transform>
            <Position>(0,0,0)</Position>
        </Transform>
        <Material>
            <Asset>sandstone</Asset>
            <NormalMap>true</NormalMap>
            <ParallaxHeightScale>0.15</ParallaxHeightScale>
        </Material>
        <AABB>
            <Offset>(7.75,-0.125,7.75)</Offset>
        </AABB>
        <Shadow>
            <CullFace>1</CullFace>
        </Shadow>
    </TerrainPatch>
</Terrain>
```

Les nouveautés sont les suivantes :

- Le noeud *Generator* spécifie un générateur à utiliser pour produire le terrain, et la seed exploitée par celui-ci. Les propriétés *Prop* de *Generator* sont essentiellement celles de l'octaveur.
- La section *HeightModifier* accueille deux nouveau modifieurs :
  - *Erosion* pour l'érosion.

  - *Offset* pour indiquer une translation du terrain (verticale uniquement pour l'instant). La sémantique initialement prévue s'en trouve clarifiée : On *génère* puis on *modifie*…

- Les objets peuvent maintenant être positionnés relativement à une heightmap déclarée comme "principale" dans un chunk, c'est le rôle de l'attribut main="true" de *TerrainPatch*. Tout *Model* ou *ModelBatch* qui possède l'attribut ypos="relative" sera repositionné verticalement selon l'altitude locale. C'est la fonction helper ground_model() du *SceneLoader* qui permet cela.

```
<Model ypos="relative">
    ...
</Model>

<ModelBatch instances="25" seed="48" ypos="relative">
    ...
</ModelBatch>
```

Erosion

L'algo d'érosion que j'utilise est adapté de WEngine (pompé à l'époque sur le net). Comme je ne l'ai pas documenté ni commenté, j'ai du mal à piger exactement comment il fonctionne sur le plan physique, à part que c'est un algo de relaxation de mesh avec une règle infléchissant la formation de talus. Cet algo forme de beaux plateaux dans un paysage et je l'aime bien, mais il faut avouer qu'un faible pourcentage de l'espace paramétrique fournit des résultats exploitables. Par ailleurs il existe d'autres façons de faire à explorer (notamment les algos d'érosion hydrodynamiques à simulation de gouttelettes) pour lesquels les résultats semblent excellents :

https://ranmantaru.com/blog/2011/10/08/water-erosion-on-heightmap-terrain/

[Material]

J'ai amélioré la classe *Material* laissée dans un piteux état par deux mois de flemme sélective. C'est propre, c'est net. Un *Material* tombe dans deux catégories : * texturé -> Utilisation de maps pour le shading (textures albedo, roughness, metallic, AO, depth, normal) * non-texturé -> Utilisation de valeurs homogènes pour le shading (tint, scalar roughness, scalar metallicity, transparency)

Celà est maintenant clairement renforcé par les deux constructeurs possibles. Des accesseurs ont été ajoutés, et la transparence est spécifiée (seuls les matériaux non texturés peuvent être transparents pour le moment). La *Scene* ne possède plus qu'une seule fonction pour ajouter un modèle, comment ce modèle est stocké en interne selon qu'il est transparent ou opaque, est géré dans la fonction Scene::add_model(). Voilà qui devrait simplifier les choses.

```
Note : Peut être qu'un peu de polymorphisme dynamique renforcerait davantage la
   sémantique texturé/non-texturé...
```

[BUG][fixed] Transparent models VBO misindexing

Les objets transparents ne s'affichent plus correctement (comme si on indexait le mauvais VBO, genre j'ai un bout de mesh de terrain au lieu d'un cube transparent), je crois qu'il faut utiliser un VAO distinct (?)… -> En effet, il a fallu introduire un vertex array spécialisé pour la géométrie transparente.

[Hermite Splines]

La classe template *CSpline* est un interpolateur à splines cubiques de Hermite. Toutes les mathématiques sont posées en détail dans le cahier, mais très simplement, une spline est une fonction polynomiale continue par morceaux, et une spline de Hermite spécifie ses polynomes sous la forme de Hermite (donc des couples (valeur,tangente)). Une spline cubique est entièrement définie par : * la donnée d'une liste de points attachés à des loci de l'espace paramètre * les deux tangentes aux extrémités (ou une règle pour les évaluer) * une police d'évaluation des tangentes internes

Les paramètres template T et Tg représentent respectivement le type interpolé (scalaire, vecteur… n'importe quoi avec une structure d'espace vectoriel ou de module sur un anneau en vérité), et la police d'évaluation des tangentes aux points de contrôle. Trois polices sont définies dans le namespace *CSplineTangentPolicy* : * *Finite* pour une évaluation type éléments finis * *CatmullRom* pour les splines de Catmull-Rom * *Cardinal* pour les splines cardinales (à mi-chemin entre une spline de Catmull-Rom et une spline aux tangentes internes uniformément nulles, selon la valeur d'un *paramètre de tension*)

A l'instar de la classe *Bezier* la classe de splines possède un état et définit une méthode d'évaluation rapide interpolate(float x) basé sur une expression en *loi de puissance* (made in Bob). Les tangentes et coefficients (du type T interpolé) sont calculés automatiquement à l'initialisation des points de contrôle. Pour relâcher la contrainte sur l'espace de paramètre d'être l'intervalle [0,1], j'utilise des transformations affines pour remapper l'espace d'entrée [t_min,t_max] vers une collection d'espaces [0,1] (un pour chaque sous-domaine de l'interpolant) sous lesquels l'expression des interpolants locaux est uniforme. La partie multiplicative de ces transformations affines est déportée dans le pré-calcul des coefficients pour plus de rapidité d'évaluation (oui, c'est de la micro-optimisation, et alors).

Une fonction dbg_sample() permet d'exporter dans un fichier log un nombre donné d'échantillons de la spline, avec la possibilité de l'évaluer hors domaine (en extrapolation). Ceci m'a permis de valider mes classes après import sous Matlab, et continue de me servir temporairement pour l'édition de telles splines (voir le script WCore/logs/-plot_splines.m).

[Daylight System]

La nouvelle classe de haut niveau *DaylightSystem* simule une alternance jour/nuit. Son action peut être interrompue et reprise via une pression sur la touche F5. Cette classe utilise des splines et un conteur float interne pour changer dynamiquement la direction de la lumière directionnelle, sa couleur, son intensité, et des paramètres de post processing tels que le vecteur gamma, la saturation, et la densité de fog. Le domaine d'interpolation est l'intervalle [0,24[. Chaque paramètre évolue selon des points de contrôle fixés à des heures particulières de la "journée". Par exemple, la densité du fog doit croître au petit matin, et s'annuler le jour venu, et est donc déterminée par une spline initialisée comme suit :

```
pp_fog_density_interpolator_({0.0f,6.0f,8.0f,22.0f,23.9f},
                             {0.02f,0.05f,0.0f,0.0f,0.02f})
```

La saturation baisse pendant la nuit, pour simuler une vision scotopique :

```
pp_saturation_interpolator_({0.0f,6.0f,18.0f,22.0f,23.9f},
                            {0.8f,1.0f,1.4f,1.0f,0.8f})
```

On pompe les rouges le soir et les bleus la nuit pour plus de "vibrance" :

```
pp_gamma_interpolator_({0.0f,6.0f,18.0f,23.9f},
                       {{1.0f,1.1f,1.2f},
                        {1.0f,1.1f,1.1f},
                        {1.2f,1.1f,1.0f},
                        {1.0f,1.1f,1.2f}})
```

Etc.

Quand il fait jour, la lumière directionnelle simule le Soleil, et la Lune quand il fait nuit. La position du Soleil et de la Lune est simulée par une trajectoire circulaire spécifiée en coordonnées sphériques (avec un angle d'inclinaison orbitale fixe, et un angle d'élévation dépendant du temps). L'idée d'incliner les trajectoires n'est pas innocente : ça permet d'éviter que la light cam ne passe par une singularité au zénith. Si tel était le cas la shadow map sursauterait et les ombres avec. Cette singularité est d'origine "mécanique", et provient du fait que ma classe de cam ne peut pas regarder exactement à la verticale (pour éviter un blocage de cardan). Le système modifie aussi la position regardée par la light cam afin de limiter la casse avec ces foutus objets qui décident de sortir de la shadow map et n'ont plus d'ombre tout à coup.

[BUG][fixed] Fresnel black disk artifact

J'avais cet étrange artéfact en disque noir qui bouffait les spécu proche de la cam, qui n'avait jamais vraiment été corrigé même après mes notes du [30-07-18]. En fait j'utilisais l'asset test "pavedFloor" pour le sol, lequel produit assez peu de Fresnel, limitant ainsi la portée de l'artéfact, mais l'effet est réapparu plus tard quand j'ai swappé pour la texture "sandstone" qui est plus brillante. Plus tard encore, j'ai observé que cet effet dépendait de la norme du vecteur position de la lumière directionnelle. En réalité, j'oubliais tout simplement de normaliser la direction source-fragment dans le shader, donc ma BRDF avait des zéros partout où ce vecteur était surunitaire. Sale con, mange tes morts.

[Optim] Gaussian Spherical Fresnel approximation

Mes tests étant devenus réellement fiables à partir du moment où j'ai lancé un système de seeds, j'ai pu mesurer avec plus de précision les coûts/bénéfices de plusieurs traits -supposés- d'optimisation (les données sont dans le cahier). En particulier, passer d'une approximation de Fresnel-Schlick à une approximation Gaussienne sphérique dans le shader lpass conduit à un gain moyen de 66µs par frame.

[Optim] LoD enabled parallax mapping

En revanche, activer le parallax mapping pour un LoD<2 n'économise que 40µs sur une frame comparativement au full parallax mapping (gros branchement dynamique qui pète le front d'onde du GPU), donc je ne me donne même pas la peine.

A garder en mémoire cependant pour devenir un vrai Shader Master 7ème dan :

```
float lod = textureQueryLod(mt.albedoTex, texCoords).y;
```

[02-09-18] Ero-Zi0n

[Terrain] Scale : Une échelle pour aller au ciel

- Le maillage des terrains a été remanié pour diminuer le cisaillement et la torsion (le sens des triangles change d'une ligne à l'autre).
- On peut contrôler un paramètre d'échelle "textureScale" pour la texture du terrain. C'est possible parce que j'ai modifié les UVs du mesh pour qu'ils évoluent sur toute la taille du mesh plutôt que [0,1], et que les textures sont en wrap par défaut.
- La densité du maillage peut être contrôlée uniformément via un autre paramètre d'échelle "latticeScale". La taille du terrain est maintenant spécifiée en mètres plutôt qu'en taille de heightmap. La taille de la heightmap est ajustée selon l'échelle, et la heightmap elle-même enregistre cette échelle, de sorte que sa fonction get_height(const vec2&) soit elle aussi spécifiée en mètres.
- L'octaveur intègre également un paramètre d'échelle. Ce paramètre, s'il n'est pas initialisé dans le XML, prend la même valeur que "latticeScale", de sorte qu'augmenter la finesse du maillage n'entraine pas une réduction de l'échelle du terrain.

Voici un exemple de la section concernée dans le XML :

```
    <TerrainPatch width="32" length="32" height="0" main="true">
        <Prop name="latticeScale">0.5</Prop>
        <Prop name="textureScale">1.2</Prop>
        <Generator type="simplex" seed="8">
            <!--<Prop name="scale">0.5</Prop>-->
            <Prop name="octaves">10</Prop>
            <Prop name="frequency">0.01</Prop>
```

```
            <Prop name="persistence">0.4</Prop>
            <Prop name="loBound">0.0</Prop>
            <Prop name="hiBound">25.0</Prop>
        </Generator>
```

Ici la densité du maillage est multipliée par 4 (x2 en X et x2 en Z), et la texture est légèrement contractée. La propriété “scale” commentée du noeud *Generator* indique la valeur par défaut, identique à “latticeScale”.

Erosion Hydro-Erosion : Un algo bien classe

J’ai transcrit l’algo de E-DOG (https://ranmantaru.com/blog/2011/10/08/water-erosion-on-heightmap-terrain/) pour l’hydro-érosion de terrain basé sur la simulation de gouttelettes. C’est de la bombe !

Il est intégré sous le nom de terrain::erode_droplets(). Son fonctionnement est assez complexe, comme il s’agit de modéliser l’écoulement d’une gouttelette d’eau selon le gradient local, la dissolution du sol (selon la pente locale, la vitesse de la gouttelette, sa quantité de solide déjà dissout…) et la redéposition du solide dissout en contrebas.

La “taille” de la gouttelette dépend en vérité de la finesse du mesh et n’est pas configurable (jusqu’à ce quelle le soit). Plus précisément, l’algo tombe dans la catégorie des systèmes particulaires, lesquelles particules ont un rayon d’action unitaire dans le maillage. Par conséquent l’algo produit des résultats qualitativement différents selon la densité de maillage. Par ailleurs, la position 2D des particules étant tirées au hasard dans le maillage, une même seed donnera également des résultats qualitativement différents selon la taille mémoire de la heightmap. C’est possible à corriger si je le souhaite, sans trop de difficulté en récupérant les positions interpolées qui elles, sont soumises à l’action de l’échelle du maillage.

Le plus gros problème que je prévois est celui des raccords avec les patchs de terrains des chunks adjacents, lorsqu’ils seront implémentés. En effet, lorsqu’une particule échappe à la heightmap, elle est détruite et on passe à la suivante. Donc elle n’aura bien sûr pas d’action sur le patch d’à côté. Donc on aura des problèmes de continuité aux bords. Le seul moyen que je vois est d’appliquer l’érosion de manière globale quand on a l’info sur tous les chunks. Pour que ça fonctionne il faudrait trouver un moyen pour que les particules puissent changer de chunks, le bordel…

Le XML a changé pour s’accomoder d’un nouveau modificateur de type érosion. Pour l’ancien algo qui fait des plateaux et des talus :

```
    <Erosion type="plateau">
        <Prop name="iterations">20</Prop>
        <Prop name="talus">0.4</Prop>
        <Prop name="fraction">0.5</Prop>
    </Erosion>
```

Pour l’algo d’hydro-érosion :

```
    <Erosion type="droplets">
        <Prop name="iterations">500</Prop>
        <Prop name="Kq">10.0</Prop>
        <Prop name="Kw">0.001</Prop>
        <Prop name="Kr">0.9</Prop>
        <Prop name="Kd">0.02</Prop>
        <Prop name="Ki">0.1</Prop>
        <Prop name="Kg">40.0</Prop>
        <Prop name="minSlope">0.05</Prop>
        <Prop name="epsilon">1e-3</Prop>
        <Prop name="seed">1</Prop>
    </Erosion>
```

[05/06-09-18] Doc à écrire

[X] Better *Shader* -> template send_uniform() -> template send_uniforms() -> REM spec func -> #include directive parsing [X] Better *InputHandler* -> parse xml file -> register action -> better debouncing [X] Better *DebugOverlayRenderer* -> register debug pane [ ] SSAO -> *SSAOBuffer* -> *SSAORenderer* [X] Better *Scene* -> single traverse_models() [X] Motion -> *ConstantRotator* -> no more hard code in *Scene* [X] w_symbols.h -> properties enum classes

[06-09-18] Better everything

J'ai apporté de nombreuses modifications au code afin d'améliorer l'interface de certaines classes qui laissait à désirer. J'ai aussi implémenté un système de SSAO (occlusion ambiante), et vidé la *Scene* de tout son contenu hard codé.

[Shader] Améliorations

Fin de l'invasion des send_uniform_X()

Une seule fonction template send_uniform() permet maintenant l'envoi d'uniforms aux shaders. Toutes les fonctions over-spécifiques comme update_uniform_MVP() ont disparu. J'ai conservé les comportements spécifiques aux lumières et matériaux (pour lesquels la structure interne définit des schémas d'envoi d'uniforms assez variés) dans des fonction send_uniforms() (noter le 's' final).

#include "yo_mamma.glsl"

Par ailleurs j'ai ajouté une fonction de parsing de directives #include dans les shaders ! OpenGL, contrairement à DX ne propose pas cette fonctionnalité (ils ont leurs raisons chez Khronos). C'est facile à implémenter côté client donc aucun problème. J'ai simplement codé une méthode privée Shader::parse_include() qui réalise le merge du code d'une inclusion donnée avec le source du shader. Je n'ai pas pris la peine de rendre cette fonction récursive, pour éviter d'avoir à coder la gestion de gardiens et autres trucs ennuyeux des préprocesseurs. Un niveau d'includes c'est suffisant pour l'instant.

Les includes portent tous l'extension .glsl (ce n'est pas une necessité) et sont localisés dans le dossier shaders/include/ (ça en revanche c'en est une).

Tout ceci implique que les numéros de lignes seront plus difficiles à tracker dans les rapports d'erreurs à la compilation GPU (à moins d'un effort de parsing des messages d'erreur…) mais en revanche mes shaders sont plus simples, mieux organisés, et moins de code est dupliqué d'un shader à l'autre.

[InputHandler] True to its name

InputHandler possède enfin une fonction handle_inputs() ! Plusieurs améliorations sont faites pour afiner le comportement des touches au debouncing, et l'organisation interne est plus claire.

Enregistrement d'actions

Les lambdas qui servaient à InputHandler::stroke_debounce() dans la fonction d'update du main initialisent maintenant des actions automatiques grâce à InputHandler::register_action() qui associe un key binding à une action. Un key binding est représenté par une structure *KeyBindingProperties* qui comporte : * le compteur cooldown pour le debouncing "cooldown" * la valeur de réinitialisation du cooldown "cooldown_reset_val" * l'évenement déclencheur "trigger" * la touche associée "key" * et un booléen "repeat" qui détermine si l'action doit être répétée si la touche est restée enfoncée.

Une telle structure est initialisée lors de l'appel à set_key_binding() et stockée dans une map qui l'associe à un "binding name" (hash string).

L'appel à register_action() associe une action (foncteur void) à chaque binding name, de sorte que la fonction handle_keybindings() peut maintenant poll tous les évenements claviers et lancer les actions associées automatiquement.

Vrai debouncing

La fonction stroke_debounce() est affinée pour laisser la possibilité à un key binding d'être répétable (l'action est répétée à intervalle régulier de durée "cooldown") ou bloquant (l'user doit relâcher la touche pour que le "cooldown" puisse descendre). Il est toujours possible d'affecter une action à un key release, c'est indépendant.

```
void InputHandler::stroke_debounce(GLFWwindow* window,
                                   hash_t binding_name,
                                   std::function<void(void)> Action)
{
    const uint16_t& key  = key_bindings_.at(binding_name).key;
    const uint16_t& trig = key_bindings_.at(binding_name).trigger;
    const bool& repeat   = key_bindings_.at(binding_name).repeat;

    auto evt = glfwGetKey(window, key);
    if(evt == trig)
    {
        if(ready(binding_name))
        {
            Action();
            hot(binding_name);
            return;
        }
        if(!repeat)
            hot(binding_name);
    }
    if(evt == GLFW_RELEASE)
    {
        cold(binding_name);
        return;
    }
    cooldown(binding_name);
}
```

Parsing XML

Le fichier res/xml/w_keybindings.xml contient une structure regroupant l'ensemble des key bindings que j'utilise pour l'instant en dev. En voilà un bout :

```
<?xml version="1.0" encoding="utf-8"?>
<KeyBindings>
    <Category name="DebugControls">
        <KB name="k_run"          key="LEFT_SHIFT"   cooldown="10"
           trigger="press"/>
        <KB name="k_walk"         key="LEFT_SHIFT"   cooldown="0"
           trigger="release"/>
        <KB name="k_forward"      key="W"/>
        <KB name="k_backward"     key="S"/>
        <KB name="k_strafe_left"  key="A"/>
```

```
        <KB name="k_strafe_right" key="D"/>
        <KB name="k_ascend"       key="SPACE"/>
        <KB name="k_descend"      key="LEFT_CONTROL"/>
        <KB name="k_tg_fog"       key="F"            cooldown="20"/>
```

La fonction InputHandler::import_key_bindings() peut parser un tel fichier et générer / enregistrer automatiquement les key bindings qui vont bien.

5 attributs sont initialisables : * “name” (**obligatoire**) est le nom du key binding, le même est utilisé par register_action(). * “key” (**obligatoire**) est le nom de la touche du clavier. Le fichier keymap.h contient une grosse map qui associe les hash des noms de touches aux valeurs utilisées par GLFW. Basiquement j'ai copié le nom des symboles de GLFW en omettant le préfixe GLFW_KEY_. C'est grâce à ça que je peux parser le nom de touche. * “cooldown” est le nombre de cycles de cooldown pour le debouncing. Défaut = 0. * “trigger” est l'évenement déclencheur de l'action (pour l'instant “press” ou bien “release”) Défaut = “press” * “repeat” qui n'est pas montré dans le listing précédent, est le booléen qui contrôle le comportement de répétition de l'action quand la touche est maintenue.

InputHandler conserve le DOM après parsing, de sorte que les key bindings puissent être remappés dynamiquement et enregistrés (plus tard).

[DebugOverlayRenderer] Moins ad hoc

La structure dbg::DebugTextureProperties enregistre un indice de texture OpenGL, un nom de sampler à afficher et un booléen qui l'identifie ou non comme une depth map (le rendu est différent pour une depth map, pour laquelle je linéarise la profondeur avant le tone mapping). Le type alias *DebugPane* est un std::vector et représente un lot de texture à afficher en bas de l'écran.

Une fonction DebugOverlayRenderer::register_debug_pane() peut enregistrer un tel lot de textures, lesquelles sont soit précisées individuellement via des vecteurs en argument. Une deuxième fonction du même nom permet d'enregistrer automatiquement toutes les textures d'un *BufferModule* en argument dans un panneau de debug.

Du coup, la fonction render() n'a plus qu'à sélectionner le panneau courant dans son vecteur de *DebugPane* pour l'afficher.

[Motion] Classe ConstantRotator

La classe nouvelle *ConstantRotator* de motion.hpp permet de faire tourner un objet avec une vitesse angulaire constante. Son interface XML est la suivante :

```
    <Motion>
        <ConstantRotator>
            <Prop name="angular_rate">(0,90,0)</Prop>
        </ConstantRotator>
    </Motion>
```

La classe elle-même prend un pModel en argument comme cible, contrairement à *PositionUpdater* qui prend une référence vers le vecteur position. Comme je n'ai encore pas de système d'entité, je ne peux pas regrouper sous une même position une lumière et un modèle, par exemple. Et donc pour qu'une lumière suive un modèle je dois spécifier deux positions et deux updaters, ce qui est assez moche. *PositionUpdater* évoluera comme *ConstantRotator* pour agir sur une entité, à terme.

[Scene]

La *Scene* doit subir un gros refactor prochainement, et son interface doit être simplifiée au maximum. Par ailleurs, plus rien ne doit y être hard-codé si j'espère répartir (en particulier) le code de motion integration dans une autre classe.

Simplification de l'interface

Toutes les fonctions traverse_models_X() (y-compris traverse_models_blend_X()) sont condensées dans une unique fonction traverse_models() à comportement paramétrable. L'ordre de parcours et la catégorie de modèle (transparent/opaque) est spécifiée en argument grâce aux types enum class de w_symbols.h. Ex :

- Pour traverser les modèles transparents dans aucun ordre particulier on fait comme ça :

```
scene.traverse_models([&](std::shared_ptr<Model> pmodel)
{
    do_stuff(pmodel);
},
Scene::DEFAULT_MODEL_EVALUATOR,
wcore::ORDER::IRRELEVANT,
wcore::MODEL_CATEGORY::TRANSPARENT);
```

Scene::DEFAULT_MODEL_EVALUATOR est un prédicat statique qui retourne toujours true.

- Pour traverser les modèles opaques dans l'ordre "front to back", comme on le fait dans *GeometryRenderer* :

```
scene_.traverse_models([&](std::shared_ptr<Model> pmodel)
{
    do_stuff(pmodel);
},
[&](std::shared_ptr<Model> pmodel) // IF predicate
{
    return check_stuff_about(pmodel);
},
wcore::ORDER::FRONT_TO_BACK);
```

Fin du hard-coding

La classe *ConstantRotator* de motion.hpp m'a permis d'encapsuler les dernières lignes hard-codées dans un updater automatique. La *Scene* possède (provisoirement) une méthode add_rotator() pour ajouter un rotateur.

[TerrainPatch]

La nouvelle classe *TerrainPatch* qui n'est pour l'instant qu'un *Model* (par héritage) regroupera un ensemble de fonctionnalités spécifiques aus terrains. La nécessité (entre autres) de conserver la heightmap sous forme de texture pour du *texture splatting* m'a poussé à spécialiser le comportement des terrains. Pour l'instant, la *Scene* héberge un unique shared pointer vers un *TerrainPatch*. La fonction traverse_models() envoie le terrain en dernier, et des info de rendu y seront jointes plus tard pour indiquer au renderer appelant des directives de rendu spécifiques.

[SSAO] Depuis le temps que j'en parle

La SSAO prend place entre la passe géométrique et la passe lumière. Elle est encore au stade expérimental et est désactivée par défaut. Pour l'activer, c'est la touche 'O' pour Occlusion.

SSAO world space : Théorie

L'idée est d'estimer une occlusion pour chaque pixel de l'écran. Un voisinage de chaque pixel sur l'écran correspond à des fragments de profondeurs différentes. De tels voisinages sont échantillonnés au moyen d'un ensemble de vecteurs (samples) pour chaque pixel de l'écran, et la contribution à l'occlusion de chaque fragment du voisinage dépend de la distance au fragment testé et de l'angle entre son vecteur position et la normale locale au fragment testé. L'occlusion totale est normalisée entre 0 et 1 et stockée dans une texture.

Comme un ensemble restreint de samples est utilisé, des artéfacts apparaissent (banding) lorsque la caméra est suffisamment proche de la surface. Ce phénomène peut être endigué en appliquant une rotation pseudo-aléatoire aux samples, au moyen d'une "normal map" qui tessèle l'écran (j'utilise une texture 64x64 générée à la volée). Du bruit fixe par rapport à l'écran peut être visible dans certaines conditions, et il peut être nécessaire d'appliquer un flou gaussien à la texture SSAO après sa génération.

Implémentation

L'algo que j'utilise est tiré de [Mendez] qui lui-même dérive d'une implémentation de Crytek. C'est une implémentation réputée rapide, immune à l'auto-occlusion (hallowing artifact) puisque non basée sur une depth map et adaptée à mon GBuffer en world space. Les voisinages ont un rayon diminuant linéairement avec la profondeur du fragment testé, afin d'adapter l'échantillonnage à la perspective. Et le nombre d'itérations est adaptatif (plus le fragment est profond, moins on itère).

```
Occlusion = max( 0.0, dot( N, V) ) * ( 1.0 / ( 1.0 + d ) )
N: occludee normal
V: occluder-occludee vector
```

Je l'ai modifié pour tenir compte de l'orientation de la normale locale par rapport à la lumière directionnelle. En effet, j'observais une occlusion trop forte et peu réaliste à des endroits pleinement exposés à la lumière, et le contraste faisait ressortir le bruit. Un simple produit scalaire vient moduler l'intensité de l'effet d'occlusion. J'ai aussi modifié le calcul d'occlusion pour le rendre plus rapide. Notamment, j'ai changé la dépendance en distance de linéaire à quadratique. L'effet d'occlusion est plus prononcé (tout en restant subtil avec le bon set de paramètres) et le calcul plus rapide (on évite une racine carrée). J'ai aussi factorisé le paramètre d'intensité pour l'appliquer à l'occlusion totale.

```
Occlusion = max( 0.0, dot( N, V) ) * ( 1.0 / ( 1.0 + d^2 ) )
```

La classe *SSAOBuffer* est le *BufferModule* (et singleton) utilisé par la classe *SSAORenderer* comme texture d'occlusion.

Le shader SSAO.vert/frag est paramétrable via les uniforms suivants (de la struct render_data) : * float f_radius; -> Rayon max des voisinages * float f_intensity; -> Intensité de l'effet * float f_scale; -> Coeff de d^2 * float f_bias; -> Biais au produit scalaire N.V

Pour appliquer l'effet d'occlusion ambiante, il existe différentes manières dont une seule est la bonne (voir [Mentalray]). Beaucoup de sources que j'ai pu lire (a priori non listées ici) indiquent que l'occlusion ambiante doit être appliquée en post processing, parfois de manière soustractive, parfois multiplicative. L'image aura alors une apparence sale, et l'effet sera très difficile à doser (je ne le sais que trop bien). L'occlusion directionnelle est déjà calculée séparément, ça s'appelle l'ombre. L'occlusion **ambiante** doit être appliquée multiplicativement sur la partie ambiante de la lumière, et nulle part ailleurs. Donc lors de la passe lighting.

Or ma contribution ambiante était très faible et hard-codée dans le shader, rendant l'effet imperceptible. Donc j'ai rendu cette contribution contrôlable à travers le paramètre scalaire lt.f_ambientStrength qui est côté client une propriété des *Light*. D'ailleurs, j'interpole la force ambiante de la lumière directionnelle pendant la journée dans *DaylightSystem*. Habituellement, la contribution ambiante est conservée à une très faible valeur afin de ne pas aplatir les reliefs de l'image. Mais avec le système de SSAO, j'observe que cette valeur peut maintenant être augmentée pour donner des scènes claires, et le contraste est préservé par l'effet d'occlusion. Donc intuitivement, l'occlusion ambiante doit être comprise comme le moyen technique mis en oeuvre pour permettre à un moteur de conserver son réalisme lors de rendus avec une composante ambiante forte. Ni plus, ni moins.

SSAORenderer renferme du code vestigial d'une tentative précédente, notamment la génération de kernel (l'ensemble de samples) qui est maintenant fixée dans le shader. Je prendrai la décision plus tard de l'enlever si je suis sûr de ne plus en avoir besoin. Néanmoins, je copie ce bout de code ici pour référence :

```
    std::uniform_real_distribution<float> rnd_f(0.0, 1.0);
    std::default_random_engine rng;

    // Generate a random hemispherical distribution of vectors in tangent space
```

```
    for (uint32_t ii=0; ii<KERNEL_SIZE_; ++ii)
    {
        // Random vector in tangent space, z is always positive (hemispherical
           constraint)
        vec3 sample(rnd_f(rng) * 2.0 - 1.0,
                    rnd_f(rng) * 2.0 - 1.0,
                    rnd_f(rng));
        sample.normalize();
        sample *= rnd_f(rng);

        // Accelerating lerp on scale to increase weight of occlusion near to the
        // fragment we're testing for SSAO
        float scale = ii/float(KERNEL_SIZE_);
        scale   = lerp(0.1f, 1.0f, scale * scale);
        sample *= scale;

        ssao_kernel_.push_back(sample);
    }
    // ...
    // Optionnel : On peut générer une texture 1D avec
    glGenTextures(1, &kernel_texture_);
    glBindTexture(GL_TEXTURE_1D, kernel_texture_);
    glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB32F, KERNEL_SIZE_, 0, GL_RGB, GL_FLOAT,
       &ssao_kernel_[0]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

Sources :


```
[Mendez] https://www.gamedev.net/articles/programming/graphics/
a-simple-and-practical-approach-to-ssao-r2753
[Chapman] http://john-chapman-graphics.blogspot.com/2013/01/ssao
-tutorial.html
[LOgl] https://learnopengl.com/Advanced-Lighting/SSAO
[LOgl2] https://learnopengl.com/
code_viewer_gh.php?code=src/5.advanced_lighting/9.ssao/9.ssao.fs
[Meteor] https://electronicmeteor.wordpress.com/2011/12/26/applying
-ssao-to-scenes/
[Codeflow] http://codeflow.org/entries/2011/oct/25/webgl-screenspace
-ambient-occlusion/
[Pyalot] https://github.com/pyalot/WebGL-City-SSAO/blob/master
/ssao/gd.shader
[Mentalray] http://mentalraytips.blogspot.com/2008/11/joy-of
-little-ambience.html
```


En pratique

C'est lent. Je ne peux pas tenir les délais en full HD (drops à 30 fps intermittents), mais ça reste acceptable en 1366x768 (+2-3 ms).

[SSS] Application détournée de la SSAO

J'ai lu des slides très intéressants montrant qu'on peut se servir d'une SSAO pour estimer une carte d'épaisseur et simuler du sub-surface scattering (SSS) pour pas cher et de manière convaincante (voir [GDC2011]). Il suffit d'inverser les normales dans le shader (j'ai déjà cette fonctionnalité dans SSAO.frag avec rd.b_invert_normals) afin de calculer une occlusion *dans* les objets translucides. Les zones les plus fines seront donc naturellement les plus occludées. Cette carte d'occlusion est ensuite inversée et appelée carte d'épaisseur. Puis on peut dérouler une SSS rapidement avec quelques paramètres light-dependant et quelques paramètres material-dependant :

```
    half3 vLTLight = vLight + vNormal * fLTDistortion;
    half fLTDot = pow(saturate(dot(vEye, -vLTLight)), iLTPower) * fLTScale;
    half3 fLT = fLightAttenuation * (fLTDot + fLTAmbient) * fLTThickness;
    outColor.rgb += cDiffuseAlbedo * cLightDiffuse * fLT;
```

Les paramètres material-dependant sont écrits dans le GBuffer lors de la passe géométrique. * per-material -> fLTAmbient front/back translucency that is always present -> iLTPower power value for direct translucency -> fLTDistortion subsurface distortion (normal displacement) -> fLTThickness thickness map * per-light -> fLTScale scale of light transport inside object

Sources :


```
[GDC2011] https://colinbarrebrisebois.com/2011/03/07/gdc-2011
-approximating-translucency-for-a-fast-cheap-and-convincing
-subsurface-scattering-look/
[Zucconi1] https://www.alanzucconi.com/2018/09/02/shader
-showcase-saturday-8/
[Zucconi2] https://www.alanzucconi.com/2017/08/30/fast-subsurface
-scattering-1/
```


[07/08-09-18] Doc à écrire / Features

- [X] Better shaders -> global #define system
- [X] Better logger -> inline multi-tag parsing (except raw/track mode) -> style / icon maps
- [X] Better render statistics -> mean / stdev / median / min / max
- [ ] Variance shadow mapping -> **EXPERIMENTAL_VARIANCE_SHADOW_MAPPING** -> p_max adjust

[08-09-18] De l'utile, de l'agréable

- Le logger a été rénové en profondeur de sorte à ne plus avoir besoin des séquences ANSI pour changer le style.
- Une amélioration mineure de la classe *Shader* rend trackable les options de compilation dans les shaders.
- Les statistiques de rendu sont plus complètes.
- Le Variance Shadow Mapping a été implémenté.

[Logger] Debugger avec style

Les styles et pseudo-icônes du *Logger* ont été placés dans des maps avec des clés de type MsgType. De fait les switch ont disparu du code. J'ai implémenté un parseur regex qui cherche dans les messages soumis à DLOGx() des tags de la forme : plop et les remplace par les séquences ANSI qui vont bien. Les tags sont toujours de simples caractères et font référence à un type d'information que l'on veut mettre en valeur à travers un style fixe. Le premier tag est toujours remplacé par un des styles de la map TAG_STYLES :

```
p : chemins d'accès    : "\033[1;38;2;0;255;255m"   : bleu clair
n : noms et symboles   : "\033[1;38;2;255;50;0m"    : orange sombre
i : instructions / op  : "\033[1;38;2;255;190;10m"  : orange clair
v : valeurs            : "\033[1;38;2;190;10;255m"  : violet
u : uniforms / attrib  : "\033[1;38;2;0;255;100m"   : vert pomme
d : valeurs par défaut : "\033[1;38;2;255;100;0m"   : orange vif
b : mauvaises choses   : "\033[1;38;2;255;0;0m"     : rouge
g : bonnes choses      : "\033[1;38;2;0;255;0m"     : vert
z : choses neutres     : "\033[1;38;2;255;255;255m" : blanc
x : noeuds XML         : "\033[1;38;2;0;206;209m"   : turquoise
```

Le deuxième tag est toujours remplacé par le style par défaut du message, afin de rétablir le style d'origine. Les messages de type RAW ou TRACK échappent au système de parsing. Le programme test_logger.cpp (target test_logger) a été rédigé pour tester les nouveaux features au fur et à mesure.

Exemples C++ :

```
    DLOGF("[Texture] Couldn't find named texture: <n>" + name + "</n>");
    DLOGI("<g>Compiled</g> vertex shader from: <p>" + vertexPath + "</p>");
    DLOGI("<i>#version</i> <v>" + glsl_version_ + "</v>");
```

C'est beaucoup plus clair et moins pète couilles à écrire que :

```
    // Grosse flemme
```

[Shader] Un système de defines trans-shader semi-automatisé

La petite fonctionnalité bien utile. *Shader* définit une map statique :

```
std::vector<std::string> Shader::global_defines_ =
{
#ifdef __EXPERIMENTAL_LIGHT_VOLUMES__
    "__EXPERIMENTAL_LIGHT_VOLUMES__",
#endif
#ifdef __EXPERIMENTAL_VARIANCE_SHADOW_MAPPING__
    "__EXPERIMENTAL_VARIANCE_SHADOW_MAPPING__",
#endif
};
```

qui contient donc des strings correspondant à des options de compilation, ssi ces options sont définies à la compilation. A la création de chaque shader, juste après le parsing de la version (nouveauté également) et juste avant les includes, une directive #define est placée dans le source pour chaque valeur de la map (méthode privée setup_defines()).

Ainsi, il suffit de rajouter un bloc ifdef dans la map pour garantir que le define pourra être utilisé tel quel dans tous les shaders et aussi dans leurs includes.

[Stats] Amélioration de la FIFO de temps de rendu

L'objet *MovingAverage* utilisé dans la main loop pour fournir des statistiques sur les temps de rendu exporte maintenant une struct *FinalStatistics* dont les membres contiennent la valeur moyenne, l'écart type, la médiane et les valeurs extrémales du temps de rendu sur N points.

[Variance Shadow Mapping] Parce que PCFSM date de 1987

[25-09-18] Doc à écrire

- [X] Program argument parser (arguments.h)
- [X] _Pipeline_
- [X] main() simplifications scene.setup_user_inputs(input_handler); pipeline.setup_user_inputs(input_handler); daylight.setup_user_inputs(input_handler);
- [ ] Variance Shadow Mapping (encore)
- [X] Chunks

[02-10-18] Ca ressemble de loin à un niveau

Déménagement à Pau terminé, on reprend le taff. Que va-t-on bien pouvoir faire sans Internet ? Pas mal de trucs, il se trouve.

[Pipeline] Roi du pétrole

La classe *Pipeline* encapsule l'ensemble des renderers et propose une interface pour modifier leurs comportements et faire un rendu. Rien d'extraordinaire, mais ça laisse libre d'implémenter quelques fioritures. Notamment, je peux profiler chaque renderer et afficher des statistiques pour chacun d'eux. C'est l'option **PROFILING_RENDERERS** qui active cette possibilité.

L'utilisation est simple :

```
    RenderPipeline pipeline(scene);
    //...
    pipeline.setup_user_inputs(input_handler);
    //...
    context._render([&]()
    {
        pipeline.render();
    });
```

Voir pipeline.h

Better main()

main() a perdu un peu de poids grâce aux dernières encapsulations. *InputHandler* est de plus en plus "self-contained". Les systèmes principaux doivent définir une méthode setup_user_input() (pour l'instant non virtuelle, il n'y a pas encore d'interface commune aux systèmes) qui prend en argument le *InputHandler* (en attendant qu'il soit singletonisé ?). Dans ces fonctions, les systèmes définissent les actions à associer aux keybindings.

Le context setup se réduit donc à :

```
    context._setup([&](GLFWwindow* window)
    {
        // Map key bindings
        scene.setup_user_inputs(input_handler);
        pipeline.setup_user_inputs(input_handler);
        daylight.setup_user_inputs(input_handler);
        input_handler.register_action(H_("k_tg_mouse_lock"), [&]()
        {
            input_handler.toggle_mouse_lock();
```

```
            context.toggle_cursor();
        });
    });
```

Par ailleurs, le parsing des arguments de main() se fait maintenant au moyen de la fonction parse_program_arguments() de arguments.h, qui initialise une structure *ProgramOptions* exploitable par *GLContext* :

```
    ProgramOptions options = parse_program_arguments(argc, argv);
    GLContext context(options);
```

[Chunks] Un gros morceau

Le refactor a dû être soigneusement préparé tant le système projette loin dans le code.

Refactor Méthode

Supposons que l'on ait une classe A dont certains membres x_i et certaines méthodes f_j() agissant sur les x_i uniquement, doivent être encapsulés dans une autre classe B, de telle sorte qu'en fin de refactoring, A dispose non pas d'une unique instance de B, mais d'une *collection* de B. L'approche que j'ai suivie intuitivement est la suivante :

1) Modifier les fonctions d'initialisation des x_i de sorte à rendre les B utilisables dans une collection. En particulier, la hiérarchie des données constituantes de B est à réfléchir à l'avance.
2) Dans le header de A, déclarer une nouvelle classe B en amont avec l'ensemble des x_i en private. Déclarer A comme classe amie de B.
3) Remplacer l'ensemble des x_i de A par une seule instance de B qu'on nommera B_0, et modifier chaque accès à un membre x_i en B_0.x_i (ou B_0->x_i si B_0 est un pointeur). Comme A est friend, elle peut accéder directement aux membres private de B_0.
4) Copier les A::f_j() dans l'interface de B. Les fonctions B::f_j() accèdent/mutent les membres B::x_i. Modifier les A::f_j() pour tirer partie des B::f_j(). Certaines A::f_j() deviennent naturellement candidats à l'inlining.
5) Après avoir vérifié que tout fonctionne, remplacer B_0 par une instance de B dans une collection Bs (vector, map…). Si nécessaire, ajouter un membre k_0 à A pour enregistrer l'unique clé ou indice de l'instance de B dans la collection Bs. Toutes les occurences de B_0.x_i sont remplacées par Bs[k_0].x_i.
6) Ajouter à l'interface de A des fonctions pour créer/initialiser/détruire des B dans la collection Bs.
7) A ce stade, l'interface de la classe A n'a toujours pas changé, et donc A est fonctionnellement identique à sa première version, du point de vue du code extérieur. Modifier alors les méthodes A::f_j() *une par une* pour inclure l'indice en argument supplémentaire. Ca va péter du code à l'extérieur, et c'est là toute l'idée. On se sert des erreurs compilo pour tracer les sections de code extérieur utilisant A qui doivent être modifiées pour s'accomoder du changement d'interface de A.
8) Bouger la classe B dans une paire de fichiers B.h et B.cpp.

Certaines méthodes A::f_j() conserveront la même signature, ou bien disparaîtront, d'autres deviendront inline, on se débrouille juste pour écrire des choses qui ont du sens.

Refactor Scene

La méthode précédente est appliquée à la classe *Scene* qui possède maintenant une collection de *Chunk*. Un coup d'oeil dans chunk.h permet d'identifier les x_i et f_j().

Dans un premier temps, j'ai dû me demander à quoi correspondait un chunk. Je suis arrivé à l'idée que pour mon jeu, un chunk est la donnée d'un bout de terrain carré de taille fixe (*TerrainChunk*), ainsi que de modèles, lumières et quelques updaters simples (*PositionUpdater*, *ConstantRotator*). Les chunks d'une map sont disposés dans une

grille et possèdent des coordonnées sous-multiples des coordonnées world : * Le chunk (0,0) est celui dont l'origine est la position monde (0,0,0). * Le chunk (0,1) a son origine en (0,0,chunk_size) * Le chunk (2,0) a son origine en (2 * chunk_size,0,0) L'origine d'un chunk est le coin inférieur/arrière/droit (x_min,y_min,z_min).

On peut par ailleurs supposer que plusieurs terrain chunks contigus sont générés avec la même seed et les mêmes propriétés de génération, mais plusieurs types de terrains peuvent coexister dans la même map (biomes/régions) avec des seed et set de paramètres différents. Un modèle simple que j'ai décidé d'implémenter est le suivant. Chaque type de terrain est déclaré dans un noeud *TerrainPatch* (**dont la sémantique a changé**), enfant du noeud *Terrain*, enfant du noeud *Scene*. J'insiste bien sur l'idée que *TerrainPatch* est maintenant un noeud global. Un **terrain patch** est une zone rectangulaire spécifiée par une origine et une taille. L'origine d'un terrain patch est la position (chunk coords) du chunk arrière-droit (de coordonnées minimales sur tout le patch). La taille du patch est spécifiée en nombre de chunks.

```
<Scene>
    <Terrain chunkSize="32" latticeScale="0.5" textureScale="0.25">
        <!-- Patch P1 -->
        <TerrainPatch origin="(0,0)" size="(2,4)" height="0">
            <Generator type="simplex" seed="6">
                <!--<Scale>0.5</Scale>-->
                <Octaves>10</Octaves>
                <Frequency>0.01</Frequency>
                <Persistence>0.4</Persistence>
                <LoBound>0.0</LoBound>
                <HiBound>30.0</HiBound>
            </Generator>
            <HeightModifier>
            <!-- ... -->
        </TerrainPatch>

        <!-- Patch P2 -->
        <TerrainPatch origin="(2,0)" size="(1,3)" height="0">
            <Generator type="simplex" seed="92">
                <!-- ... -->
        </TerrainPatch>
        <!-- ... -->
    </Terrain>
    <Chunk coords="(0,0)">
        <Models>
            <!-- ... -->
    </Chunk>
    <Chunk coords="(0,1)"/>
    <Chunk coords="(0,2)"/>
    <Chunk coords="(0,3)"/>
    <!--<Chunk coords="(1,0)"/>-->
    <Chunk coords="(1,1)"/>
    <Chunk coords="(1,2)"/>
    <Chunk coords="(1,3)"/>
    <Chunk coords="(2,0)"/>
    <!--<Chunk coords="(2,1)"/>-->
    <Chunk coords="(2,2)"/>

    <!-- Orphan chunk == ERROR -->
    <Chunk coords="(3,4)"/>
    <!-- ... -->
```

```
                    ^ z
[oo].[  ].[  ].[  ] |
[  ].[  ].[P1].[P1] |       [P1] généré par terrain patch 1
```

```
[  ].[P2].[P1].[P1] |       [P2] généré par terrain patch 2
[  ].[--].[P1].[P1] |       [  ] void chunk
[  ].[P2].[--].[P1] |       [--] silent chunk
x <-----------------+       [oo] orphan chunk (ERROR)
```

SceneLoader::parse_patches() va dans un premier temps parser les noeuds *TerrainPatch* et générer une carte d'association des chunks aux patchs. Une table des chunks est aussi tenue à jour à chaque fois qu'un noeud *Chunk* est rencontré.

- [**Silent Chunk**] Pour qu'un chunk puisse être chargé, il **doit** être déclaré avec un noeud *Chunk* dans le fichier map, sinon il est ignoré (silent chunk), même si un patch le recouvre. En théorie, rien n'empêche un patch de surcharger un silent chunk d'un patch précédent. Cela déclenche néanmoins un warning quand **DEBUG_CHUNK** est défini.
- [**Orphan Chunk**] Un chunk qui serait déclaré sans appartenir à un patch est orphelin (orphan chunk). Les chunks orphelins sont *interdits* et génèrent une erreur.
- [**Void Chunk**] Un chunk non recouvert par un patch et non déclaré n'est simplement pas un chunk (void chunk).

Comme un chunk est identifié par ses coordonnées, une clé peut être calculée à partir de celles-ci par hashing. Les coordonnées chunk utilisent des math::i32vec2 et j'ai défini une fonction std::hash pour ce type. Dans la *Scene*, un *Chunk* est identifié au moyen d'une telle clé.

Un chunk est toujours *lazy initialized*. Le *SceneLoader* avec sa méthode load_chunk() qui peut être appelée dynamiquement, demande dans un premier temps à la scène de construire un chunk, puis parse à la volée le terrain, les modèles, les batches et les lumières afin d'initialiser le chunk nouvellement alloué, et d'envoyer la géométrie à OpenGL :

```
    scene.add_chunk(chunk_coords);
    parse_terrain(scene, chunk_coords);
    parse_models(scene, chunk_node, chunk_index);
    parse_model_batches(scene, chunk_node, chunk_index);
    parse_lights(scene, chunk_node, chunk_index);
    scene.load_geometry(chunk_index);
```

En définissant l'option **PROFILING_CHUNKS** un timer *nanoClock* réalise le profiling de chacun des appels précédents.

Sans aucune optimisation, le temps de chargement d'un chunk scale sans surprise en fonction du contenu :

Pour crystal_scene.xml (chunks 64x64 = 32mx32m). Chunk (1, 1): 10 crystal instances

```
[*]   [SceneLoader] Loading chunk: 352683992 at (1, 1)
   -->    Chunk loaded in 9604.58 µs total.
   -->    Terrain: 5883.09 µs
   -->    Models: 0.189 µs
   -->    Model Batches: 306.038 µs
   -->    Lights: 0.17 µs
   -->    Geometry upload: 3415.09 µs
```

Chunk (0, 2): 750 crystal instances

```
[*]   [SceneLoader] Loading chunk: 612114758 at (0, 2)
       -->    Chunk loaded in 22152.3 µs total.
       -->    Terrain: 5734.43 µs
       -->    Models: 0.214 µs
       -->    Model Batches: 10447.7 µs
       -->    Lights: 0.207 µs
       -->    Geometry upload: 5969.75 µs
```

Le timing du chargement de terrain peut être largement optimisé. A noter que tout ça c'est du temps CPU, potentiellement sur un fil d'exécution à part (quand j'aurai un *ChunkManager* multi-threadé). On est souvent sous la frame en ordre de grandeur, donc il est déjà envisageable d'utiliser ce système en temps-réel.

Un chunk peut être supprimé dynamiquement au moyen de Scene::remove_chunk().

```
    // in main.cpp (TMP)
    // Load a single patch (several chunks)
    uint32_t xmax = 3,
             ymax = 4;

    for(uint32_t xx=0; xx<xmax; ++xx)
        for(uint32_t yy=0; yy<ymax; ++yy)
            scene_loader.load_chunk(scene, math::i32vec2(xx, yy));

    // Remove chunk at (1,1)
    scene.remove_chunk(math::i32vec2(1, 1));
```

Alternativement, on peut passer une clé (position hash) à remove_chunk(). De plus, load_chunk() retourne le hash de position du chunk qui vient d'être loadé :

```
    uint32_t chunk_index = scene_loader.load_chunk(scene, math::i32vec2(1, 1));
    // Remove chunk at (1,1)
    scene.remove_chunk(chunk_index);
```

La *Scene* est le seul objet capable de retrouver les coordonnées d'un chunk à partir de sa clé, grâce à un simple lookup via l'accesseur :

```
    inline const math::i32vec2& Scene::get_chunk_coordinates(uint32_t chunk_index)
       const;
```

Les fonctions sort_x() et traverse_x() de la *Scene* itèrent sur les chunks (eux-même classés par ordre de distance dans la scène) et exécutent les fonction Chunk::sort_x() et Chunk::traverse_x() pour chaque chunk. Ainsi, la notion de chunk est complètement interne à la classe *Scene* et le monde extérieur continue de traverser la scène comme avant, à l'exception près que les visiteurs de Scene::traverse_x() prennent un argument uint32_t supplémentaire qui est la clé du chunk parent du modèle / de la light visité :

```
    // in geometry_renderer.cpp -> render()
    scene_.traverse_models([&](std::shared_ptr<Model> pmodel, uint32_t chunk_index)
    {
        scene_.bind_vertex_array(chunk_index);
        // ...
        scene_.draw(pmodel->get_mesh().get_n_elements(),
                    pmodel->get_mesh().get_buffer_offset(),
                    chunk_index);
        // ...
    },
    /* MODEL EVALUATOR PREDICATE */,
    wcore::ORDER::FRONT_TO_BACK);
```

C'est un peu convolué, mais j'entends absorber tout ça dans traverse_x() un peu plus tard. Ca évitera si je m'y prends bien d'avoir à bind/unbind sans arrêt dans une boucle de rendu…

Manque plus qu'une classe *ChunkManager* pour superviser le chargement/déchargement des chunks dans la boucle d'updates en fonction de la position de la caméra principale.

[07-10-18] Doc à écrire

- [ ] Variance Shadow Mapping (encore, encore)

- [ ] Better scene traversal -> Scene::draw_models() -> Chunk frustum culling
- [ ] *ChunkManager*
- [ ] Better DaylightSystem -> CSpline parsing -> Ad Hoc mais fonctionnel
- [ ] Ad Hoc shadow virtual cam lookat model -> Les ombres ne sont plus trop dans les fraises et suivent vaguement la cam.
- [ ] gfx_driver wrapper
- [x] _TreeGenerator_ -> Spline trees -> Spline skinning -> Tentacles

[12-10-18] Spline l'Ancien

Pas mal d'améliorations, un peu de nouveauté.

[TreeGenerator] Comment faire un arbre avec des splines

J'ai eu cette idée, probablement aux chiottes (étant toujours sans Internet, fallait bien que je sorte ça de mon cul), de faire des arbres en procédural à partir de spline skinning. Pour l'occasion, j'ai créé une nouvelle scène (tree_scene.xml) et j'ai commencé à bosser sur une hiérarchie de splines dans un algo procédural de la classe statique *TreeGenerator*. Le principe est relativement simple. * On commence avec un tronc qui est une spline globalement verticale et on appelle celui-ci une "branche" de niveau 0. * On va ensuite parcourir le tronc en itérant sur le paramètre de la spline, et à chaque pas, la branche a une probabilité non nulle de faire un noeud. * Pour chaque noeud de la branche on crée des branches filles de niveau 1 (la probabilité de créer une branche est un paramètre du modèle), qui partent dans une direction comprise entre une orthogonale aléatoire à la branche 0 et la direction de la branche 0 (un paramètre d'angle sert à lerp entre ces deux vecteurs). * On répète l'algo sur les branches de niveau 1 nouvellement crées, de sorte à produire des branches de niveau 2, etc. On arrête l'algo quand le niveau maximal est atteint.

Mon algo est récursif, et le niveau mentionné précédemment n'est autre que la profondeur de récursion. Il est implémenté dans la fonction make_spline_tree_recursive() (pour l'instant statique dans tree_generator.cpp).

Une fois toutes ces splines générées, il m'a fallu implémenter un moyen commode de les afficher. J'ai donc créé un nouveau type de noeud de la scène : les *LineModel*. Ca se déclare basiquement comme un modèle, mais ça génère des Mesh et des primitives GL_LINES. Essentiellement, j'ai dû rajouter un *BufferUnit* et un *VertexArray* dans chaque *Chunk* pour contenir de tels objets, et une fonction pour traverser et dessiner de tels modèles. Il y a une forte chance pour que ce système ne serve qu'à afficher de la géométrie de debug pour longtemps (peut être que le futur HUD system en tirera partie), quoi qu'il en soit, le rendu est optionnel et activé/désactivé au moyen d'une pression sur la touche 'K'.

Basiquement, je sample chaque spline et crée des segments de courbes que j'ajoute au mesh, c'est ce que fait la fonction TreeGenerator::generate_spline_tree(). C'est le *DebugRenderer* qui pour l'instant va traverser de tels objets dans la *Scene* pour les dessiner. L'affichage est un peu buggé maintenant à cause de modifications ultérieures, et je réparerai tout ça en temps voulu. Mais ce système m'a permi de tester l'algo et de le raffiner quelque peu.

[Spline Skinning] Tentacules

Une fois satisfait par la hiérarchie de splines, il m'a fallu écrire une fonction pour faire du **spline skinning**. L'idée est d'habiller une spline au moyen de tronçons de cylindres dont le rayon peut évoluer avec le paramètre de la spline (j'utilise une loi de puissance, l'exposant est le paramètre de contrôle). Basiquement, il s'agit de générer un tentacule !

La fonction factory::skin_spline() prend en arguments un *Mesh* et une spline Catmull-Rom sur vec3 (alias *math::CSplineCatmullV3*) plus quelques paramètres de contrôle (nombre de sections, nombre d'échantillons, rayon initial et exposant d'évolution du rayon). Le but de l'algo de skinning est de calculer des cercles régulièrement espacés, tels que la spline est orthogonale aux plans de tous ces cercles en leurs centres.

Pour chaque section, on calcule les tangentes b_y et t_y à la spline au début et à la fin de la section, nécessitant l'échantillonnage de la spline en 3 points successifs A, B et C.

```
    for(uint32_t rr=0; rr<nRings-1; ++rr)
    {
        // Compute 3 levels of circle centers (along spline)
        float t0 = rr*step;
        float t1 = (rr+1)*step;
        float t2 = (rr+2)*step;
        A = spline.interpolate(t0);
        B = spline.interpolate(t1);
        C = spline.interpolate(t2);
        // ...
    }
```

Les tangentes locales en A et B sont données respectivement par :

```
b_y = B-A
t_y = C-B
```

Deux bases locales sont calculées en début et fin de section pour générer des cercles correctement orientés. Pour ce faire, le produit vectoriel du vecteur arbitraire (1,0,0) (*A AMELIORER*) par les tangentes donne les éléments en z, et un produit vectoriel des tangentes avec les éléments en z donne les éléments en x.

```
xx := (1,0,0)
b_z = normalize(xx  ^ b_y)
b_x = normalize(b_y ^ b_z)
t_z = normalize(xx  ^ t_y)
t_x = normalize(t_y ^ t_z)
```

Les directions locales en x et z sont orthogonales à la spline et entre-elles et peuvent donc servir de base pour la génération de cercles :

```
C_A : {A + R_A*cos(theta)*b_x + R_A*sin(theta)*b_z | theta \in [0,2*\pi]}
C_B : {B + R_B*cos(theta)*t_x + R_B*sin(theta)*t_z | theta \in [0,2*\pi]}
```

A partir de là c'est fastoche (du moins, autant que du meshing peut l'être). On termine en ajoutant une section de cône à la fin de la spline.

La fonction skin_spline() ne fait qu'ajouter des vertices et des triangles à un modèle, mais ne calcule pas les normales ni les tangentes. Donc elle peut être appelée plusieurs fois sur un même modèle avec des splines différentes, de sorte que le modèle final regroupe les mesh associées à chaque spline.

La fonction factory::make_tentacle() produit un *Mesh* au moyen d'une unique spline, afin de tester l'algo de skinning. Un nouveau type de mesh ("tentacle") est créé dans la scène pour l'occasion :

```
    <Model ypos="relative">
        <Mesh>tentacle</Mesh>
        <Material>
            <Color>(0.5,0.2,0)</Color>
            <Roughness>0.2</Roughness>
        </Material>
        <Transform>
            <Position>(10,0.0,20)</Position>
            <Scale>5.0</Scale>
            <Angle>(5,0,-20)</Angle>
        </Transform>
    </Model>
```

La spline est hard-codée dans le *SceneLoader* pour le moment, c'était juste un test, mais je vois l'intérêt d'avoir une primitive tentaculaire dans un moteur de jeu, donc ça sera paramétrable à l'avenir.

[Spline Tree Skinning]

Une fois qu'on a notre fonction de spline skinning, il suffit de l'appliquer en série sur chaque spline de la hiérarchie d'un arbre. Il m'a fallu rajouter une liste de rayons en argument de make_spline_tree_recursive(), de sorte que la hiérarchie transporte également un rayon de branche. Cette liste est mise à jour symmétriquement par rapport à la liste de splines lors de la construction de l'arbre. La fonction TreeGenerator::generate_tree() fait très exactement ça. Le skinning est adaptatif, et le nombre de sections et d'échantillons circulaires dépend des dimensions de la branche. En pratique j'utilise le rayon initial centré réduit comme estimateur de ces dimensions :

```
    // Iterate over splines to generate segments
    for(uint32_t jj=0; jj<splines.size(); ++jj)
    {
        float rr = (radii[jj]-rmin) / (rmax-rmin); // between 0 and 1
        uint32_t n_samples  = (uint32_t) math::lerp((float)props.min_samples,
           (float)props.max_samples, rr);
        uint32_t n_sections = (uint32_t) math::lerp((float)props.min_sections,
           (float)props.max_sections, rr);
        factory::skin_spline(pmesh, splines[jj], n_sections, n_samples, radii[jj],
           props.radius_exponent);
    }
```

Ainsi, les branches les plus grosses (en particulier le tronc) auront une géométrie plus détaillée, et inversement pour les plus petites.

Une fois le mesh construit, les normales et tangentes sont construites et lissées.

[Paramétrisation]

Voici un exemple d'arbre dans la scène :

```
    <Model ypos="relative">
        <Mesh>tree</Mesh>
        <TreeGenerator>
            <Seed>8</Seed>
            <Recursion>2</Recursion>
            <NodeProbability>0.7</NodeProbability>
            <BranchProbability>0.7</BranchProbability>
            <Twist>0.02</Twist>
            <MaxBranch>2</MaxBranch>
            <BranchAngle>0.3</BranchAngle>
            <BranchHindrance>0.8</BranchHindrance>
            <ScaleExponent>1.5</ScaleExponent>
            <RadiusExponent>0.8</RadiusExponent>
            <TrunkRadius>0.1</TrunkRadius>
            <MinSamples>4</MinSamples>
            <MaxSamples>10</MaxSamples>
            <MinSections>5</MinSections>
            <MaxSections>10</MaxSections>
            <MaxNodes>8</MaxNodes>
        </TreeGenerator>
        <Material>
            <Color>(0.5,0.2,0)</Color>
            <Roughness>0.8</Roughness>
        </Material>
        <Transform>
            <Position>(35,-0.5,20)</Position>
            <Scale>15.0</Scale>
```

```
        </Transform>
    </Model>
```

C'est un peu brouillon comme hiérarchie, mais ça fait le job. Quand un noeud *Mesh* a la valeur "tree", alors un noeud *TreeGenerator* **DOIT** être présent (en sibling) et éventuellement définir un certain nombre de propriétés (pour lesquelles je précise un intervalle de valeurs "workable") :

Seed -> Seed du générateur pseudo-aléatoire. *Recursion* -> Niveau de récursion, entre 1 et 4, jamais plus, sinon explosion combinatoire. *MaxNodes* -> Nombre maximum de noeuds le long d'une branche. 5 à 8 donnent des résultats exploitables. *NodeProbability* -> Probabilité de création d'un noeud. De 0.3 à 0.8 disons. *BranchProbability* -> Probabilité de création d'une branche sur un noeud. 0.6, 0.7, dans ces eaux-là. *Twist* -> Tortuosité de l'arbre. C'est l'amplitude de déviation aléatoire d'une branche par rapport à sa direction générale. De 0 à 0.2. *MaxBranch* -> Nombre maximal de branches par noeud. De 2 à 5. *BranchAngle* -> A quel point une branche fille s'écarte de la direction de sa mère. Entre 0 (suit sa mère) et 1 (ortho). *BranchHindrance* -> Facteur d'encombrement stérique des branches. De 0.0 à 1.0 (respectivement, répartition complètement aléatoire à répartition complètement symétrique) *ScaleExponent* -> Exposant d'échelle pour la taille des branches. De 1.0 à 3.0. *RadiusExponent* -> Exposant d'échelle pour l'évolution du rayon des branches. De 0.3 à 1.0. *TrunkRadius* -> Rayon initial du tronc. De 0.05 à 0.3 j'imagine, la taille globale de l'arbre est plus volontiers fixée par le noeud *Transform.Scale*. *MinSamples* -> Nombre minimal d'échantillons circulaires. 3 est une valeur sûre, 2 fonctionne également. *MaxSamples* -> Nombre maximal d'échantillons circulaires. Selon la taille physique du tronc entre 10 et 20 pour un modèle détaillé. Je ne mettrais pas en dessous de 5-6. *MinSections* -> Nombre minimal de sections pour les splines. 2, 3, 4. *MaxSections* -> Nombre maximal de sections pour les splines. Entre 5 et 20.

Remarques subsidiaires : * Quelques fonctionnalités servent à enjoliver la génération, par exemple, des branches filles ne sont jamais créées près de l'origine de la branche mère, et toujours créées à la fin de celle-ci. * [X] Très certainement que le modèle va évoluer. En particulier j'aimerais rendre contrôlable la symétrie de l'arbre. Les branches filles prennent pour l'instant des directions d'azimuth aléatoire, et il est donc possible d'avoir genre 4 branches qui partent dans des directions proches, ce qui n'est pas du meilleur effet. Un paramètre d'anisotropie peut aussi être sympa. -> *BranchHindrance* * L'arbre peut bien sûr être texturé (le parallax mapping déclenche un freeze, cependant, donc se contenter du normal mapping pour le moment).

Quoi qu'il en soit, je peux déjà générer des arbres de styles très différents, et reconnaissables en tant que tels (en tant qu'arbres morts, du moins, je n'ai pas encore bossé sur les feuilles). Je suis assez rarement content de moi, là c'est le cas.

[14/15-10-18] TODO

[o] Dans terrain_patch.cpp le constructeur de TerrainChunk peut utiliser factory::make_terrain_tri_mesh() au lieu de factory::make_terrain() pour construire un terrain en triangle mesh pur (= 4 fois moins de vertices). Cependant c'est 3-4 fois plus long à charger. Causes probables : -> pmesh->build_per_vertex_normals_and_tangents(); -> N'utilise pas encore de triangle classes (à l'instar des position classes) -> Permettrait un traverse_equal_range au lieu d'un double for -> création des vertices et triangles en 2 passes

Il faut faire le calcul des normales et tangentes locales sur place.

Ce qui serait génial par ailleurs serait de pouvoir demander les normales locales à la fonction de bruit directement. Je sais qu'il est possible de calculer un gradient analytiquement pour un bruit de simplex…

```
J'ai ajouté une HashMap qui associe à chaque position (et donc à chaque vertex) la
    liste des triangles (des offsets vers les premiers indices de triangles dans
    indices_) qui contiennent le vertex en question. Je peux donc traverser des
    "triangle classes" à l'instar des position classes. La fonction
    pmesh->build_per_vertex_normals_and_tangents() est 4 fois plus rapide, et le
    temps de chargement d'un terrain est redevenu acceptable, même avec un calcul
    ultérieur des normales/tangentes.
```

[ ] Une nouvelle distinction apparaît qui devrait être renforcée dans le code de *Mesh*. La distinction est d'évocation artistique au départ, mais implique des comportements et des cas d'utilisation différents de la classe *Mesh*. * Les

meshes pour lesquels on veut voir des faces et des angles, par nécessité de la multiplicité des normales et tangentes en chaque point anguleux, possèdent plusieurs vertices à la même position. La construction automatisée des vertices et des normales sur de tels meshs se fait *par triangle* et repose sur l'utilisation d'une HashMap qui enregistre les classes de position, de sorte que l'ensemble des vertices à une position donnée peut être accédé rapidement. * Les meshes pour lesquels on veut avoir une impression de courbure lisse, jusque-là utilisaient la même structure de donnée que le type de mesh précédent, mais l'appel à une fonction de smoothing permettait d'en altérer l'apparence. Lors de mes récentes optimisations pour baisser le coût de vertex de mes scènes, de tels meshs ont été remplacées par de véritables triangle meshes (en représentation shared vertex). Pour de tels meshes, les vertices sont tous à des positions différentes, mais peuvent intervenir dans plusieurs triangles. La construction automatisée des normales et tangentes se fait alors *par vertex* et repose sur l'utilisation d'une autre HashMap qui enregistre les classes d'appartenance des vertices à leurs triangles, de sorte que l'ensemble des triangles référençant un vertex soit accessible rapidement.

Pour l'instant, ces deux types de fonctionnement sont gérés par des ensembles distincts de fonctions dans la classe *Mesh*. Mais le code me supplie de polymorphiser tout ça. On pourrait imaginer une classe *MeshWalker* et deux dérivées *FaceMeshWalker* et *TriangleMeshWalker*. Ces classes implémenteraient les deux politiques respectives et seraient composées à la classe *Mesh*, qui choisirait l'instanciation à la construction.

Avantages : -> Empreinte mémoire plus faible. -> On ne serait plus obligé d'avoir 2 HashMaps mais 1 seule. -> La HashMap pourrait être détruite avec le *MeshWalker* une fois qu'on n'en a plus besoin. -> Suffisamment peu critique pour que le polymorphisme dynamique ne représente pas d'overhead majeur. -> Peut être étendu facilement à d'autres comportements. -> Pour le cas, par exemple, où un mesh provient d'un .obj qui précise d'ores et déjà les normales et tangentes, un null object *NullMeshWalker* implémenterait une politique -à la Hollande- qui consisterait à ne rien faire. -> Interface plus simple (plus à choisir entre Mesh::build_per_vertex_x() et Mesh::build_x()).

[15-10-18] Doc à écrire

- [ ] Variance Shadow Mapping (encore, encore, encore)
- [o] Better scene traversal -> Scene::draw_models() -> Chunk frustum culling
- [ ] *ChunkManager*
- [o] Better DaylightSystem -> CSpline parsing -> Ad Hoc mais fonctionnel
- [o] Ad Hoc shadow virtual cam lookat model -> Les ombres ne sont plus trop dans les fraises et suivent vaguement la cam.
- [o] gfx_driver wrapper
- [ ] Vertex opts -> Terrain hex triangle mesh -> #vertex/4 -> moins de torsion -> toujours aussi rapide grâce à Mesh::traveres_triangle_class()
- [ ] Icosahèdres, icosphères et subdivision de triangles.
- [ ] *RockGenerator* -> Déformation d'une icosphère par bruit de simplex périodique.
- [ ] *Config* singleton -> Templated get<>(), set<>() -> recursive xml parsing
- [x] Shadow mapping, orthographic frustum thight fit.

[18-10-18] Bosser sous speed...

[Shadow Mapping] Orthographic frustum tight fit optimization

Enfin !! Mes ombres sont enfin beaucoup moins merdiques. L'idée (et ce que je refusais de faire jusqu'alors par peur de devenir dingue) est de se servir des vertices du view frustum (*FrustumBox*) de la *Camera* principale afin de déterminer les limites du frustum orthographique de la cam virtuelle représentant la lumière directionnelle. Les vertices sont calculées automatiquement par la fonction update() de *FrustumBox*. Il convient alors de les transformer dans l'espace lumière par multiplication à gauche par la *matrice de vue* de la light cam une fois positionnée. En effet, il faut pouvoir estimer les limites (xmin, xmax, ymin, ymax, zmin, zmax) dans l'espace lumière (l'axe z correspond à la direction de vue lookAt), pour contraindre le frustum orthographique à épouser au mieux le view frustum. La matrice de vue de la light cam réalise précisément la transformation world->light space.

```
1 Rappel : pour une camera, View^-1 = Model et Model^-1 = View
```

J'ai tout fourré dans le *DaylightSystem* comme un gros porc, parceque j'avais le scope parfait pour prototyper le bouzin, mais c'est pas si crétin que ça au final, que ce système prépare le terrain pour la génération de la shadowmap directionnelle. Donc ça va probablement rester là-dedans modulo un peu de réarrangement/optimisation. Voilà comment je procède :

1) Après avoir updaté la position de la lumière directionnelle (juste après le calcul orbital), j'update la position de la light cam et je lui demande de regarder vers la position de la cam principale (anciennement dans setup_camera()). L'appel à Camera::look_at() force un update de sa *FrustumBox* en interne. Donc à ce stade j'ai mes vertices en world space (récupérables grâce à Camera::get_frustum_corners()) et une view matrix de précalculée.

2) Chaque vertex est converti en light space via une multiplication à gauche par la view matrix de la cam principale. A ce stade, il ne reste plus qu'à rechercher les coordonnées extrémales dans l'ensemble des vertices en light space. On obtient les bornes xmin, xmax, ymin, ymax, zmin et zmax. En théorie on peut donc initialiser une matrice de projection ortho avec ces données pour la light cam (mais on ne va pas le faire tout de suite). Une nouvelle fonction Camera::set_orthographic(float[6]) permet en effet d'initialiser un frustum ortho à partir de telles bornes dans un tableau de float (les éléments au nombre de 6, sont rangés dans l'ordre sus-mentionné). Cependant les bornes ne sont pas utilisables en tant que telles. zmin et zmax sont pour l'instant fixées en dur (à -10 resp. 500-1000) pour éviter des problèmes de clipping near/far. Je n'ai pas encore trouvé de bonne stratégie pour gérer ces deux-là, a priori ça dépend surtout de l'AABB de la scène toute entière, donc c'est pas si simple.

3) Un peu de redimensionnement est donc de mise. Je "zoom" artificiellement en divisant les bornes en x et y par 3.5, mais ce que je devrais véritablement faire, c'est une renormalization des bornes par la longueur de la grande diagonale du view frustum. Celà permettrait de s'assurer que le view frustum tient toujours entièrement dans le shadow frustum. A venir prochaînement.

4) Un artéfact très désagréable est alors visible lors de mouvements de caméra : un sintillement des ombres dû à de l'aliasing. Il suffit de renormaliser les bornes en x et y à nouveau, de manière à les arrondir au pixel le plus proche. Ainsi la shadow map ne sera jamais à cheval entre 2 pixels et le flickering disparaît.

5) C'est alors seulement qu'on génère une proj ortho pour la light cam.

Dans la série "perte de temps débile"

Important à souligner, en pensant initialiser mes variables de recherche de maximum correctement avec std::numeric_limits::min(), je me suis retrouvé avec des bugs hyper étranges de collapse du frustum ortho (disparition des ombres selon l'orientation de la cam) lorsque tous les y des vertices sont tous négatifs. En effet, std::numeric_limits::min() est le nombre le plus petit en *MODULE* et non en valeur absolue comme je le pensais. De fait, le ymax restait initialisé à une valeur incorrecte… Voilà comment on fait, et pas autrement :

```
static void compute_extent(const std::vector<math::vec3>& vertices, float
    extension[6])
{
    extension[0] = std::numeric_limits<float>::max();
    extension[1] = -std::numeric_limits<float>::max();
    extension[2] = std::numeric_limits<float>::max();
    extension[3] = -std::numeric_limits<float>::max();
    extension[4] = std::numeric_limits<float>::max();
    extension[5] = -std::numeric_limits<float>::max();

    for(uint32_t ii=0; ii<8; ++ii)
    {
        // OBB vertices
        const vec3& vertex = vertices[ii];

        if(vertex.x() < extension[0]) extension[0] = vertex.x();
        if(vertex.x() > extension[1]) extension[1] = vertex.x();
        if(vertex.y() < extension[2]) extension[2] = vertex.y();
```

```
        if(vertex.y() > extension[3]) extension[3] = vertex.y();
        if(vertex.z() < extension[4]) extension[4] = vertex.z();
        if(vertex.z() > extension[5]) extension[5] = vertex.z();
    }
}
```

Une fonction identique était mal codée dans bounding_boxes.cpp… J'ai donc dû corriger un bug quelque part.

Yep, après relecture de *Mesh* je me suis rendu compte que j'y avais commis la même erreur.

[19-10-18]

[Point Light Attenuation]

J'ai changé le modèle d'atténuation pour mettre un truc utilisable à la place (!) :

```
float attenuate(float distance, float radius, float compression=1.0f)
{
    return pow(smoothstep(radius, 0, distance), compression);
}
```

Et puis c'est mare. Le paramètre de compression sert à contrôler le falloff. Mes lumières ont retrouvé leur éclat…

[TODO]

- [x] REFACTOR *Mesh* -> Mesh spécialisées (triangular, shared vertices, TIN…) par héritage depuis un Mesh
- [ ] Générer des TIN directement depuis la fonction de bruit. [ ] Probablement qu'il faudra implémenter la dérivée analytique de la fonction de bruit également, de manière à

[20-10-18]

[Seamless Terrain]

En particulier depuis l'optimisation des terrain meshes, deux chunks voisins qui partagent un ensemble de positions à leur bord commun, auront tendance à y définir des normales légèrement différentes, ce qui produit une couture visible.

Un petit ajout dans SceneLoader::parse_terrain() répare les normales/tangentes au chargement des chunks. Quand un chunk est chargé, on regarde quels sont ses voisins (nord, sud, est, ouest) qui sont chargés, et pour chaque voisin on itère sur le bord commun du chunk à charger pour assigner à ses normales celles du voisin.

```
Voisin au:    Modifier le bord:    Avec le bord voisin:
sud           sud                  nord
nord          nord                 sud
est           est                  ouest
ouest         ouest                est
```

Les fonctions TerrainChunk::north(), south(), east() et west() permettent d'accéder rapidement aux bords correspondants du mesh au moyen d'un unique index en argument. La *Scene* définit une fonction traverse_loaded_neighbor_chunks() qui simplifie l'implémentation.

Les bords des chunks sont maintenant seamless.

L'algo est condensé dans la fonction terrain::fix_terrain_edges(terrain_, chunk_index, chunk_size_, scene); déclarée dans terrain_patch.h.

[22-10-18] Petite session Mesh et Shadow Mapping

Refactor Mesh

Les mesh de surface spécialisées (avec les hashmaps qui vont bien) sont maintenant dérivées de *SurfaceMesh* qui est un type alias pour *Mesh*. Les *FaceMesh* updatent des classes de position et sont optimisées pour les split vertices / modèles avec facettes apparentes, et les *TriangularMesh* sont des mesh à vertices partagé avec des classes de triangles. Les fonctions de construction des normales et tangentes sont maintenant virtuelles, plus d'emmerde, plus de confusion. Voir surface_mesh.h/cpp.

[Shadow Mapping] Optimisations

- Dans daylight.cpp : le zmax du frustum ortho de la light cam est fixé à y_cam+10, ce qui donne de bons résultats en top view.
- Le shader de shadowmapping utilise maintenant whadowmap.vert en VS au lieu de null.vert. J'introduit du *scaled normal offset* dans ce shader pour combattre l'acnée. Le principe est de déplacer les vertices dans le VS shadowmap le long de la normale locale, pour forcer les fragments dans la lumière, l'effet étant modulé par l'angle lumière/normale pour maximiser l'effet quand la surface est parallèle à la lumière, là où l'acnée est constatée. Je n'arrive pas encore complètement à contrôler ce machin, mais ça a l'air d'avoir une action…

[24-10-18]

Reconstruction de la position depuis le depth buffer

J'ai un algo pour reconstruire facilement (1 MAD) la position *en view space* depuis le depth buffer et les rayons aux coins du view frustum (précalculés en view space à chaque changement de matrice de projection *perspective*, passés en *attribut* du quad screen et donc *interpolés* au FS). Donc il faut que je puisse faire les choses suivantes :

[x] Faire mes calculs de lumière/ombres en view space, dans un premier temps avec un GBuffer position en view space. [x] Multiplier la position world par la view matrix dans le VS de la passe géométrique. [x] Les normales doivent aussi être calculées en view space. -> Multiplier les normales en model space par la transposée de l'inverse de la matrice model-view (ou juste la matrice model-view en l'occurrence, car le scaling est uniforme). [x] Il faut donner au shader light pass la position des lumières en view space [x] Pour les lumières ponctuelles il faut multiplier leur position à gauche par la matrice view (qui va donc les translater) [x] Pour la lumière directionnelle, a priori il faudra juste multiplier la "position" (direction) par la sous-matrice de rotation de la matrice view. [x] La direction de vue viewDir devient simplement -fragPos puisque la position de la cam est 0 en view space. [x] Le calcul de la position du fragment en light space : `c vec4 fragLightSpace = m4_LightSpace * vec4(fragPos, 1.0f);` devra être modifié. m4_LightSpace devra être post-multipliée par l'inverse de la view matrix. Ou bien on trouve un moyen dès maintenant de virer cette horrible multiplication matricielle du fragment shader (mal barré).

- Ca valait la peine de planifier, tout a fonctionné du premier coup, gros one shot. On a maintenant un GBuffer en view space, halle-fucking-lujah. -> Ce qui m'a beaucoup aidé, c'est d'avoir écrit mes fonctions de calcul de la lumière et des ombres de manière indépendante de la base. Tant que les paramètres d'entrée sont tous exprimés dans la même base, ça fonctionne.

[ ] Envoyer en attribut les rayons. -> Remplacer le quad 3P dans LightingRenderer::load_geometry() par un 3P2U par exemple et stocker les rayons à la création du quad dans un premier temps (on va considérer que la perspective ne bouge pas pour l'instant). 2 composantes suffisent car les rayons sont z-normalisés. Leurs composantes en z et w sont dont toujours égales à 1. [ ] Implémenter l'algo. [ ] Le VS de la light pass reçoit donc les rayons en même temps que les positions des coins du screen quad et out ces derniers dans un vec2. [ ] Le FS de la light pass calcule le paramère z_view : Si projection perspective : depth = sample_depth_buffer() z_ndc = 2 * depth+1 z_view = - M_43/(z_ndc + M_33) Sinon : … z_view = - (z_ndc * M_44 - M_43) / (z_ndc * M_34 - M_33) [ ] Puis calcule la position en view space depuis le rayon interpolé D : fragPos = z_view * D

-> **Aïe**. Dans la light pass, la partie qui traite les lumières ponctuelles n'utilise pas le screen quad comme géométrie proxy mais des sphères… Donc on ne peut pas remonter les D au VS en l'état.

[BUG][fixed] State Leak (screen texture)

Quand je shunte le FS lpass avec

```
void main()
{
    out_color = vec3(0.3,0.3,0.3);
    out_bright_color = vec3(0);
    return;
    // ...
```

au lieu de voir une couleur grise à l'écran, je vois affiché en gris les bouts de la scène qui sont dans les sphères de lumière. Désactiver le geometry renderer élimine cet effet (surement indirectement).

-> Pas un bug. Le FS n'est appelé que pour les fragments qui passent le stencil test, donc nécessairement j'observe l'empreinte de la géométrie éclairée, même si le FS sort une couleur unique… -> En revanche je m'étais bel et bien planté sur les units à bind depuis l'introduction de depthTex dans le GBuffer. C'est corrigé.

[27-10-18] UI, UI, UIIIIII

Immediate mode

ImGui est un GUI en mode immédiat (*immediate mode*). Ne pas confondre avec immediate draw call. Mode immédiat, par opposition à mode retenu (*retained mode*) signifie que le GUI est dessiné à chaque frame de manière procédurale, par des appels de fonctions. En termes MVC, l'application doit fournir une fonction qui génère une Vue à la volée en fonction de l'état présent du Modèle. Les widgets ne sont donc pas des objets, et sont construits à chaque frame. Ce type de GUI est donc *stateless* (et anti-OOP), ce qui évite les nombreux désagréments liés à la synchronisation état client / état GUI. En réalité le GUI conserve bien un état interne, mais cet état ne recouvre pas les données client, qui elles sont fournies à chaque appel. A chaque frame, les méthodes d'ImGui provoquent le remplissage d'un vertex buffer et d'une liste de draw calls, lesquels sont ensuite traités par un renderer (le mien, ou bien un renderer spécialisé de l'API). L'overhead est étonnamment faible (tant qu'on a pas de souci de fillrate), les draw batches générés à la volée sont suffisamment optimisés et cachées.

Un autre avantage lié au paradigme, est le fort couplage entre la déclaration d'un widget et son code de réaction. On mélange la logique et la présentation (et on prétend que c'est souhaitable). On peut déclarer un bouton en même temps qu'on gère sa façon de répondre avec :

```
    if(ImGui::Button("Click Me"))
        std::cout << "Please, stop, it tickles." << std::endl;
```

Y a pas photo quand on compare avec une implémentation signal/slots à la QT…

Je compte utiliser ce GUI pour le debug et l'éditeur uniquement, afin d'économiser du temps. L'UI du jeu sera en retained mode (précisément parce que la séparation présentation / logique devient importante et que je souhaite gérer cette partie en OOP).

Intégration

Intégration de Dear ImGui terminée, c'était hyper simple. D'abord, il faut configurer le loader opengl (on a le choix entre gl3w, glew et glad). Pour cela, il faut ajouter la ligne suivante dans le header imconfig.h :

```
    #define IMGUI_IMPL_OPENGL_LOADER_GLEW
```

GLContext est modifiée pour initialiser le GUI à la création de la fenêtre GLFW :

```
    // Setup Dear ImGui binding
    IMGUI_CHECKVERSION();
    ImGui::CreateContext();
    ImGui_ImplGlfw_InitForOpenGL(window_, true);
    ImGui_ImplOpenGL3_Init("#version 400 core");
    // Setup style
    ImGui::StyleColorsDark();
```

Et son destructeur est également modifié :

```
    GLContext::~GLContext()
    {
        // Shutdown ImGUI
        ImGui_ImplOpenGL3_Shutdown();
        ImGui_ImplGlfw_Shutdown();
        ImGui::DestroyContext();
        // Close OpenGL window and terminate GLFW
        glfwDestroyWindow(window_);
        glfwTerminate();
    }
```

Ensuite il suffit de signaler le début de frame à ImGui avec :

```
    // Start the Dear ImGui frame
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();
    ImGui::NewFrame();
```

Ces lignes sont placées où on veut dans la boucle, tant que toutes les lignes de génération d'UI sont appelées après.

A la fin de la boucle de rendu on dessine le GUI avec :

```
    ImGui::Render();
    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
```

Les fonctions commençant par "ImGui_Impl" proviennent des headers "imgui/imgui_impl_glfw.h" et "imgui/imgui_impl_opengl3.h". Ces fonctions sont plus ou moins données comme exemple pour le binding GLFW/OpenGL3, mais sont utilisables en tant qu'API. Ces fonctions ne sont exposées qu'à *GLContext*. Le reste de la codebase n'a qu'à inclure "imgui.h" pour pouvoir générer le GUI.

Remarques : * ImGui génère un fichier imgui.ini à la destruction du programme. Les positions des fenêtres ainsi que les tailles et états "collapsed" sont rendus persistants grâce à ça.

Plan d'action

Il semblerait qu'on soit chaud pour du design d'éditeur. Il me faut les fonctionnalités suivantes :

[x] Activation/Désactivation de certaines parties du rendu : [x] SSAO [x] Bloom [x] Forward [x] Lighting (default to albedo) [x] Shadow mapping [x] Debug display [x] AABB [x] Light proxy geometry [x] Wireframe [x] Debug overlay [x] Contrôle du rendu [x] Post processing [x] Fog [x] Enabled [x] Color [x] Density [x] Tone mapping exposure [x] Gamma [x] Saturation [ ] Contrôle de la scène [x] Contrôle du *DaylightSystem* [x] Choix de l'heure de la journée [x] Choix de l'inclinaison du soleil [ ] Contrôle du temps [ ] Vitesse de défilement [ ] Pause/Reprise [ ] Frame par frame [ ] Contrôle du système de chunks [ ] Reload chunks / map [ ] Rayon de visibilité [ ] Contrôle des objets de la scène [ ] Création [ ] Destruction [ ] Modification [ ] Repositionnement [ ] Réorientation [ ] Propriétés [ ] Sauvegarde des modifications [ ] Undo / Redo

La fonction RenderPipeline::generate_rendering_widget() se charge de générer une fenêtre avec les contrôles relatifs au rendu. J'ai fait un truc assez chiadé. La plupart des contrôles debug pertinents à cette catégorie sont accessibles

depuis l'UI, et bien plus encore. Chaque partie de la pipeline peut être activée/désactivée proprement : * Le lighting peut être désactivé, la lighting pass se borne alors à afficher l'albédo. * Le shadow mapping peut être activé indépendamment. * Il en va de même pour la SSAO, l'effet bloom et la passe forward.

Tous les affichages debug (AABB, light proxy, wireframe, debug overlay) sont accessibles depuis cette fenêtre. Les réglages du post-processing sont plus intéressants. On peut modifier en temps réel la correction gamma avec 3 sliders, la saturation, l'exposition, activer/ désactiver le fog et changer sa couleur et sa densité. La FXAA peut être contrôlée en tout ou rien pour l'instant.

Et classe internationnale : on peut afficher les temps de rendu pour chaque renderer dans des plots en temps réel, quand **PROFILING_RENDERERS** est défini.

Les différents contrôles sont répartis dans des catégories contractiles (collapsible headers).

La fonction dbg::LOG.generate_log_widget() génère un widget affichant les logs (sortie console) avec possibilité de filtrer. Simple Ctrl+C Ctrl+V depuis https://github.com/ocornut/imgui/issues/300. Une struct statique *LogWidget* est (salement) déclarée et définie dans le logger.cpp pour encapsuler la génération de fenêtre.

Remarque importante sur l'initialisation des états du GUI

Si l'on déclare par exemple une fenêtre comme suit :

```
    ImGui::SetNextWindowPos(ImVec2(10.0f, 10.0f));
2   ImGui::Begin("Rendering options");
```

Alors la fenêtre ne pourra jamais être déplacée, car sa position est reset à chaque appel de SetNextWindowPos(). Le dev d'ImGui a prévu le coup, et a introduit un deuxième paramètre enum à cette fonction comme condition d'exécution. La condition ImGuiCond_Once permet un réglage initial relaxé ensuite :

```
    ImGui::SetNextWindowPos(ImVec2(10.0f, 10.0f), ImGuiCond_Once);
2   ImGui::Begin("Rendering options");
```

Il en va de même pour l'état initial ouvert/fermé des headers contractable. Ces derniers sont des tree nodes, et peuvent donc être ouverts/fermés via SetNextTreeNodeOpen() :

```
    ImGui::SetNextTreeNodeOpen(true, ImGuiCond_Once);
2   if(ImGui::CollapsingHeader("Pipeline control"))
    {
4       // ...
```

Et oublier ImGuiCond_Once bloquerait le header dans son état initial.

Bug ImGui :

On ne peut pas mettre (en tout cas dans la même fenêtre) deux ImGui::Checkbox() avec le même label. Sinon, le deuxième ne fonctionnera pas. Probablement que le label est hashé en interne pour servir d'index dans une map quelconque, et réutiliser le même label entraîne une collision dans la map… -> En effet : https://github.com/ocornut/imgui/issues/96

[Apitrace] Magie noire

J'ai dl et compilé le programme Apitrace, qui permet de tracer de nombreuses API graphiques :

```
>> apitrace trace --api [gl|egl|d3d7|d3d8|d3d9|dxgi] /path/to/application [args...]
```

Donc pour moi :

```
1 >> apitrace trace --api gl ../bin/wcore
```

Là, un gros fichier (attention ça se remplit très vite) est généré et tous les calls OpenGl sont loggés.

On peut faire un dump texte avec :

```
>> apitrace dump wcore.trace
```

Sinon, et c'est là qu'on voit que je suis assez peu subtilement attiré par toutes les choses qui brillent, on peut faire :

```
>> apitrace replay wcore.trace
```

Et le programme va effectivement jouer un **replay graphique** de ce qui a été loggé !

Par ailleurs, le programme qapitrace (version GUI) permet une exploration détaillée de tous les calls, rangés frame par frame. On peut notamment observer les textures, les framebuffers, les shaders (source et assembleur)… Du très très lourd !

```
>> qapitrace wcore.trace
```

On peut d'ailleurs profiler le programme depuis qapitrace. Et en le faisant, je me suis rendu compte que mon blit framebuffer de la light pass prend un temps GPU DINGUE (autant qu'un draw call, voire plus). Il **faut** que je trouve un autre moyen.

Il existe un fork nommé frameretrace [3] dont j'ai vu une demo en conf qui pousse le délire encore plus loin : les shaders capturés par la trace sont modifiables, les uniforms également, et un histogramme des temps de chaque call (avec filtrage possible) permet de trouver rapidement quelles sont les frames les plus longues à rendre, et pour quelles raisons elles le sont (une modification des shaders affecte les temps des frames qui sont recalculés). On peut également faire du diffing pour identifier graphiquement l'impact d'un draw call. Je n'ai pas réussi à m'en servir à ce jour : la compilation fonctionne, mais le programme demande en dynamique des composants QT vieux comme la pierre que mon install ne semble pourtant pas pouvoir fournir…

Sources:

```
[1] https://www.khronos.org/opengl/wiki/Debug_Output
[2] https://www.khronos.org/opengl/wiki/Debugging_Tools
[3] https://github.com/janesma/apitrace/wiki/frameretrace-branch
```

[29-10-18] Opti-zonions

[Améliorations]

- La *Scene* ne possède plus les variables de taille d'écran. Le fichier globals.h déclare le singleton *GlobalData* et son alias *GLB*. parse_program_arguments() est modifiée pour écrire dans les membres de *GLB* (en accès public), et tous les systèmes qui ont besoin de la taille de l'écran n'ont qu'à inclure globals.h et se servir de GLB.SCR_W et GLB.SCR_H. Ainsi, cet état est centralisé et n'est plus dupliqué par aucun système à la construction (comme le faisaient plusieurs renderers). Ca facilitera le boulot si je dois gérer le redimensionnement de la fenêtre en temps réel…
- Les méthodes Scene::get_directional_light() et Scene::get_directional_light_nc() renvoient maintenant des weak_ptr sur la lumière directionnelle, et il incombe aux systèmes qui en ont besoin de les cast en shared_ptr via std::weak_ptr::lock(). En modifiant les systèmes dépendant de la lumière directionnelle de manière à ce qu'ils vérifient toujours que cette ressource est accessible :

```
if(auto dir_light = scene_.get_directional_light_nc().lock())
{
    // ...
```

  le moteur est maintenant stable avec une *Scene* vide (avant, c'était segfault). Et la lumière directionnelle devient a priori optionnelle. Les pointeurs sur caméra vont rester shared, car la relation de composition avec la *Scene* assure que ces ressources sont toujours accessibles tant que la *Scene* existe.

- La méthode Camera::set_orthographic_tight_fit() fait maintenant le travail de tight fitting du shadow frustum sur le view frustum, anciennement dans DaylightSystem::update(). Il faut lui passer en argument une autre caméra (la freecam), une direction de vue (la "position" du Soleil) et optionnellement les dimensions d'un texel de shadowmap si on veut faire l'arrondi au texel le plus proche pour le shadow mapping. Cette fonction est maintenant appelée dans Scene::update(). *Daylight* a été modifiée pour ne plus toucher aucune caméra de la scène.
- La *Scene* est maintenant un singleton, avec l'alias *SCENE*. Tous les systèmes qui sauvegardaient une référence à la scène (renderers, *SceneLoader*, *Daylight*, …) ont été modifiés afin d'en tenir compte. Pour l'instant ça fait sens, j'espère ne pas avoir à le regretter. J'ai commenté son interface, et virer des choses inutiles, comme les (u)nbind_vertex_array() et draw(). D'ailleurs on unbind plus, c'est inutile car il suffit de rebind par dessus…
- J'ai corrigé une fuite mémoire dans *FrameBuffer*. Après avoir passé le tableau draw_buffers_ en membre, j'ai oublié de corriger la ligne de l'allocation et j'avais :

```
GLenum* draw_buffers_ = new GLenum[n_textures_];
```

  au lieu de :

```
draw_buffers_ = new GLenum[n_textures_];
```

  De fait, je déclarais une nouvelle variable locale draw_buffers_ qui masquait le membre. Mon delete[] draw_buffers_ du destructeur agissait donc toujours sur un nullptr. Ceci m'a permis de comprendre pourquoi la fonction FrameBuffer::rebind_draw_buffers() ne fonctionnait pas. Elle est maintenant utilisée dans *LightingRenderer* pour restaurer l'écriture dans les color buffers du *LBuffer* :

```
// Stencil pass
GFX::lock_color_buffer();
// ...
lbuffer.rebind_draw_buffers();
// Light pass
// ...
```

  Et je me suis débarassé de l'horrible hack GFX::rebind_draw_buffers_2().

Le program est leak free, à part la fuite de 72bytes de x11 causée par glfw 3.1 (le bug est corrigé en 3.2 par XkbFreeKeyboard(desc, 0, True), voir [1]).

- Un petit passage sous callgrind m'apprend que AABB::update() est un CPU bottleneck en devenir (16% temps CPU). En effet, TOUS les modèles updataient leur AABB à chaque appel à Model::get_AABB(), même ceux dont on sait à l'avance qu'ils sont statiques. J'ai donc ajouté un flag bool is_dynamic_ dans *Model* mis à false par défaut. L'AABB n'est updaté lors d'un get_AABB() que si le modèle est dynamique. Pour qu'un modèle soit dynamique il faut appeler sa fonction Model::set_dynamic(true). Dans le *SceneLoader* la fonction parse_motion() pour les modèles se charge en cas de succès du parsing de passer le modèle concerné en dynamique. Toutes les fonctions du *SceneLoader* qui génèrent un *Model* ou un *TerrainChunk* doivent appeler leur fonction update_AABB() manuellement après avoir initialisé sa *Transform*.
- *GLContext* possède une liste de fonctions destinées à générer les widgets de l'éditeur. Cette liste est exécutée juste après le callback d'update si le rendu de l'éditeur est activé (évitant l'appel coûteux en CPU à ImGui_ImplOpenGL3_RenderDrawData() quand c'est inutile). Ce rendu est commuté via GLContext::toggle_editor_GUI_rendering(). Pour ajouter une fonction de génération de widget, il faut appeler GLContext::add_editor_widget_generator() avec un lambda wrapper en argument :

```
context.add_editor_widget_generator([&](){ dbg::LOG.generate_log_widget(); });
context.add_editor_widget_generator([&](){ pipeline.generate_rendering_widget();
    });
```

- La classe *Shader* n'utilise plus glGetUniformLocation() à CHAQUE putain d'appel à send_uniform(). Je me suis enfin décidé à faire ça proprement. *Shader* a comme nouveau membre une map qui associe le hash d'un nom d'uniform à sa localisation. La méthode Shader::setup_uniform_map() se charge d'initialiser cette map après le linking, en parcourant tous les uniforms actifs. Toutes les méthodes Shader::send_uniform() ont été

modifiées pour prendre un hash_t en argument au lieu d'un const char* et tous les renderers ont incorporé cette modification. Résultat, environ 40% de calls OpenGL en moins selon apitrace.

Sources:

```
[1] https://github.com/glfw/glfw/pull/662
```

[30-10-18]

[Bug] HeightmapGenerator fail in target RelWithDebInfo

```
wcore: /home/ndx/Desktop/WCore/source/src/heightmap_generator.cpp:61: static void
    HeightmapGenerator::heightmap_from_simplex_noise(HeightMap &, const
    SimplexNoiseProps &): Assertion `hm.get_width()%2==0 && "HeightmapGenerator:
    Width must be even."' failed.
Aborted (core dumped)
```

[Améliorations]

- La classe nouvelle *GameClock* encapsule les fonctions de manipulation du temps in-game, avant prototypées dans le main(). Les objets updatables accèptent maintenant un const GameClock& au lieu d'un float dt en argument de leurs fonctions update(). Ces dernières sont héritées d'une interface *Updatable*. Les *Updatable* sont stockés dans une liste et itérés lors de la phase d'update.

[01-11-18]

[Améliorations]

- J'ai apporté quelques améliorations mineures à l'ECS et espère l'intégrer bientôt. Je continue de unit tester le système. Note importante : Il est possible que l'initialisation des registres statiques via la macro REGISTER_COMPONENT soit optimisée par le compilo si le moteur est compilé en lib statique (et donc les composants ne seraient pas enregistrés). Donc soit il faudra écrire une méthode init() appelée de l'exécutable qui touche les registres afin qu'ils ne soient pas optimisés, soit :

  Use the -all_load linker option to load all members of static libraries. Or for a specific library, use -force_load path_to_archive.

  voir [1] et [2].

- Sortir du GUI ne bouge plus la caméra. Le curseur est rencentré automatiquement pour éviter le sursaut.
- L'éditeur est désactivable en définissant **DISABLE_EDITOR**.

Sources :

```
  [1] https://gamedev.stackexchange.com/
2 questions/37813/variables-in-static-library-are-never-initialized-why
  [2] https://stackoverflow.com/questions/12602513/c-executing-functions
4 -when-a-static-library-is-loaded/18678224#18678224
```

[02-11-18] C++ Black Magic

En regardant une conf d'Allan Deutsch [5] sur le dev d'API pour les jeux AAA je suis tombé sur de nouveaux idiomes dont un m'a particulièrement tapé dans l'oeil : le *detection idiom* qui permet de vérifier si une classe/structure

donnée définit une méthode de signature donnée. Le résultat de la requête est constexpr, et donc la vérification peut se faire en compile time. L'idée de base consiste à utiliser l'idiome SFINAE afin de retourner un type std::false_type/std::true_type selon qu'une substitution template foire ou pas. En gros, on vérifie la "compilabilité" d'un bout de code appelant la méthode en question.

```
namespace detail
{
template <class Default, class AlwaysVoid,
          template<class...> class Op, class... Args>
struct detector
{
    using value_t = std::false_type;
    using type = Default;
};

template <class Default, template<class...> class Op, class... Args>
struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
{
    // Note that std::void_t is a C++17 feature
    using value_t = std::true_type;
    using type = Op<Args...>;
};

struct nonesuch
{
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

} // namespace detail

template <template<class...> class Op, class... Args>
using is_detected = typename detail::detector<detail::nonesuch, void, Op,
   Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<detail::nonesuch, void, Op,
   Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;
```

Si l'on suppose que l'on a la structure suivante :

```
    struct A
    {
        void nop();
        void update(float);
        void process(int, int);

        int foo;
        float bar;
    };
```

et que l'on veuille détecter la fonction nop(), on peut définir un détecteur de la façon suivante :

```
    template <typename T>
    using nop_detector = decltype(std::declval<T&>( ).nop());
    template <typename T>
    using has_nop = is_detected<nop_detector, T>;
    template <typename T>
    constexpr bool has_nop_v = has_nop<T>::value;
```

et utiliser l'un de ces trois appels au choix :

```
    has_nop_v<A>        // by value
    has_nop<A>()        // implicit bool cast
    has_nop<A>::value   // detector value member access
```

qui renverra true pour la structure A qui déclare effectivement une méthode nop.

rq : decltype renvoie le type d'une valeur en argument. Il faut donc lui fournir une valeur, et std::declval sert justement à ça. declval sert en gros à dire "supposont que j'ai une valeur de tel ou tel type".

Si maintenant l'on veut tester la présence d'une méthode avec une signature comme process(int, int), il faut utiliser des templates variadiques :

```
template <typename T, typename... Args>
using process_detector = decltype(std::declval<T&>(
   ).process(std::declval<Args>()...));
template <typename T, typename... Args>
using has_process = is_detected<process_detector, T, Args...>;
template <typename T, typename... Args>
constexpr bool has_process_v = has_process<T, Args...>::value
```

et utiliser l'un de ces trois appels au choix :

```
    has_process_v<A, int, int>
    has_process<A, int, int>()
    has_process<A, int, int>::value
```

Les détecteurs peuvent être générés automatiquement via une macro :

```
#define GENERATE_HAS_MEMBER( FN )                                              \
template <typename T, typename... Args>                                        \
using FN## _detector = decltype(std::declval<T&>( ).FN(std::declval<Args>()...)); \
template <typename T, typename... Args>                                        \
using has_## FN = is_detected<FN## _detector, T, Args...>;
   \
template <typename T, typename... Args>                                        \
constexpr bool has_## FN## _v = has_## FN<T, Args...>::value
```

Puis pour générer un détecteur pour un nom de fonction donné, mettons process, il suffit d'appeler la macro :

```
GENERATE_HAS_MEMBER(process);
```

Et les symboles has_process et has_process_v sont générés automatiquement. Cette macro fonctionne également pour les fonctions sans argument.

On n'est pas contraint d'utiliser les types primitifs, bien entendu, et tout fonctionne correctement avec des qualifiers :

```
GENERATE_HAS_MEMBER(age_of_the_captain);

struct Foo
{
    // ...
```

```
};

struct B
{
    void age_of_the_captain(float, const Foo&);

    int baz;
    double qux;
};

int main(int argc, char const *argv[])
{
    std::cout << std::boolalpha
              << "struct 'B' has 'age_of_the_captain(float, const Foo&)': "
              << has_age_of_the_captain_v<B, float, const Foo&>
              << std::endl;
    return 0;
}
```

-> struct 'B' has 'age_of_the_captain(float, const Foo&)': true

La détection fonctionne sur des méthodes héritées. Un cas particulier intéressant est celui des déclarations ambigues du type A et B définissent toutes les deux une fonction ambiguous() et C: A,B (C hérite de A ET B) se retrouve avec un symbole ambigu dont l'appel provoque l'erreur de compilation suivante : -> error: member 'ambiguous' found in multiple base classes of different types

Eh bien, à la détection, has_ambiguous_v vaut false ! Exactement comme si on vérifiait la capacité du code "C c; c.ambiguous()" à compiler.

L'intérêt de cet idiome, est qu'une lib peut vérifier si un composant *utilisateur* hérité d'une structure de l'API déclare ou non une méthode. L'utilisateur est libre de définir ou non telle ou telle fonction, et la lib va processer le composant automatiquement en fonction de ce qu'il contient. Par exemple, la lib WCore pourrait vérifier si un composant utilisateur hérité de *WComponent* définit une méthode 'update(float dt)' et intégrer automatiquement l'appel à cette fonction dans un système d'update.

L'alias is_detected existe dans les headers expérimentaux de C++17 (std::experimental::is_detected dans) mais j'ai préféré refaire une implémentation en me basant sur [2] et [3] : voir source/sandbox/detector_idiom.cpp. Mon code reprend différents cas d'utilisation, notamment avec des fonctions virtuelles (pure, override, final). Le système ne s'étend pas aux fonctions template, mais faut pas trop en demander non plus.

Des implémentations plus anciennes comme [4] exploitent des types tags de tailles différentes et abusent de l'opérateur sizeof en lieu et place des type traits std::true/false_type.

Utilisation pratique

SFINAE et std::enable_if

Une structure *RigSFINAE* montre comment tirer partie des détecteurs pour définir un comportement selon qu'une méthode est présente ou non dans une classe :

```
GENERATE_HAS_MEMBER(update);

struct CmpBase
{
    CmpBase(char name): name_(name){}
    char name_;
};

struct CmpNotUpdatable: public CmpBase
```

```
{
    CmpNotUpdatable(char name): CmpBase(name){}
};

struct CmpUpdatable: public CmpBase
{
    CmpUpdatable(char name): CmpBase(name){}

    void update(float dt)
    {
        t_ += dt;
        std::cout << "t_= " << t_;
    }

    float t_ = 0.f;
};

template <typename Component = CmpBase>
struct RigSFINAE
{
    template<class K = Component>
    typename std::enable_if<has_update_v<K, float>, bool>::type update(K& cmp)
    {
        std::cout << "Updating component '" << cmp.name_ << "': ";
        cmp.update(0.1f);
        std::cout << std::endl;
        return true;
    }

    template<class K = Component>
    typename std::enable_if<!has_update_v<K, float>, bool>::type update(K& cmp)
    {
        std::cout << "Component '" << cmp.name_ << "' has no update(float)
           function and will not be updated." << std::endl;
        return false;
    }
};

int main(int argc, char const *argv[])
{
    RigSFINAE<> component_updater;
    CmpNotUpdatable a('a');
    CmpUpdatable b('b');

    for(int ii=0; ii<3; ++ii)
    {
        component_updater.update(a);
        component_updater.update(b);
    }

    return 0;
}
```

```
../bin/sandbox/detector_idiom
Component 'a' has no update(float) function and will not be updated.
Updating component 'b': t_= 0.1
Component 'a' has no update(float) function and will not be updated.
```

```
Updating component 'b': t_= 0.2
Component 'a' has no update(float) function and will not be updated.
Updating component 'b': t_= 0.3
```

Tag dispatching

Une autre possibilité plus simple est d'utiliser l'alias detector::value_t (qui évalue à std::true_type ou std::false_type) accessible via has_update() pour faire du tag dispatching (voir [6]) :

```
namespace detail
{
template <typename Component>
void update_dispatch(Component& component, std::false_type)
{
    std::cout << "Component '" << component.name_ << "' has no update(float)
       function and will not be updated." << std::endl;
}

template <typename Component>
void update_dispatch(Component& component, std::true_type)
{
    std::cout << "Updating component '" << component.name_ << "': ";
    component.update(0.1f);
    std::cout << std::endl;
}
} // namespace detail

template <typename Component>
void Update(Component& component)
{
    detail::update_dispatch(component, has_update<Component, float>());
}

int main(int argc, char const *argv[])
{
    CmpNotUpdatable a('a');
    CmpUpdatable b('b');

    for(int ii=0; ii<3; ++ii)
    {
        Update(a);
        Update(b);
    }

    return 0;
}
```

La sortie est rigoureusement la même.

Sources :

```
[1] https://blog.tartanllama.xyz/detection-idiom/
[2] https://en.cppreference.com/w/cpp/experimental/is_detected
[3] https://en.cppreference.com/w/cpp/experimental/nonesuch
[4] https://en.wikibooks.org/wiki/More_C++_Idioms/Member_Detector
```

```
[5] https://www.youtube.com/watch?v=W3ViIBnTTKA
[6] https://www.boost.org/community
/generic_programming.html#tag_dispatching
```

[04-11-18]

- Dans bounding_box.cpp le "HACK" de recalage des AABB s'est avéré non nécessaire suite aux récentes modifications et a donc été retiré. Tout fonctionne correctement, plus aucun objet ne disparaît à l'écran.
- Une classe *OBB* a été implémentée et ajoutée comme membre à *Model*. Le frustum culling utilise maintenant les OBBs (plus rapide). Les AABB seront reservés aux tests de collision. Les OBBs sont visualisables au même titre que les AABBs en pressant 'B' pour changer de mode d'affichage (comme pour la géométrie proxy des lumières) ou depuis l'UI.
- Les bounding boxes des terrain chunks sont maintenant recalées par défaut, plus besoin d'écrire un node *AABB* ad hoc dans les *TerrainPatch*.

[Position reconstruction]

J'ai quasiment terminé la reconstruction de la position depuis la profondeur ! Mon implémentation se base essentiellement sur [1] et mes notes. Voilà en l'état comment je fais :

lighting_renderer.cpp

```
    const math::mat4& P = SCENE.get_camera()->get_projection_matrix();
    math::vec4 proj_params(P(0,0), P(1,1), P(2,2), P(2,3));

    // in light pass
    lighting_pass_shader_.send_uniform(H_("rd.v4_proj_params"), proj_params);
```

lpass_exp.frag

```
    vec3 reconstruct_position(in float p_depth, in vec2 p_ndc, in vec4
       p_projParams)
    {
        float depth = p_depth * 2.0f - 1.0f;
        float viewDepth = p_projParams.w / (depth + p_projParams.z);
        return vec3((p_ndc * viewDepth) / p_projParams.xy, -viewDepth);
    }

    // in main()
    float depth = texture(depthTex, texCoord).r;
    vec2 pos_ndc = texCoord.xy*2.0f - 1.0f;
    vec3 fragPos = reconstruct_position(depth, pos_ndc, rd.v4_proj_params);
```

Donc en somme : 1) On sample la depth map à la position clip texCoord -> depth 2) On convertit la profondeur en coordonnées NDC -> depth_ndc = 2 * depth + 1 3) On convertit la position clip en NDC -> pos_ndc = 2 * texCoord + 1 4) La profondeur en view space est déprojetée -> depth_view = P_43 / (depth_ndc + P_33) 5) Les composantes x et y en view space sont déprojetées -> x_view = (pos_ndc.x * depth_view) / P_00 -> y_view = (pos_ndc.y * depth_view) / P_11 -> z_view = -depth_view

Le truc un peu branlant que j'ai dû faire (et ce n'est pas sans rappeler les autres trucs branlants que j'ai dû faire avec l'axe z, du fait de ma mauvaise gestion du système de coordonées lefty de GL (impossible pour moi de m'y faire, c'est pas naturel)), c'est un z-flip. Basiquement, la formule communément admise pour 4) est -> depth_view = - P_43 / (depth_ndc + P_33) et à l'étape 5) -> z_view = depth_view

Tout fonctionne à l'exception d'un petit glitch : un rectangle noir peut se former sous certaines conditions d'orientation de la cam et du Soleil, dans le ciel uniquement. Ca ressemble à une singularité de la profondeur, pas hyper étonnant en vrai. On va voir ce qu'on peut faire.

La reconstruction est activable via l'option de compilation **EXPERIMENTAL_POS_RECONSTRUCTION**. Ceci change le système en profondeur, le *GBuffer* notamment change de gueule : "positionTex" disparaît, la roughness part dans la composante alpha de l'albedo, l'overlay disparaît et le wireframe est mixé avec l'albedo plutôt qu'appliqué en overlay. Noter que la SSAO ne fonctionne plus avec cette option et est donc désactivée. Je dois déjà optimiser un max la reconstruction (en foutant un max de calculs dans le VS) avant d'intégrer cette modification pour de bon.

Cette reconstruction, certes sous-optimale, en plus de libérer 4 * 4 * 1920 * 1080 octets (31.6 Mo) de VRAM accélère déjà le rendu de 764µs en moyenne (soit 8.5%) selon mes tests. Donc il est vraiment capital que j'en vienne complètement à bout.

Sources :

```
[1] http://bassser.tumblr.com/post/11626074256
/reconstructing-position-from-depth-buffer
```

[05-11-18]

[Position reconstruction]

Afin d'intégrer le calcul des view rays dans le VS, je dois pouvoir différencier la géométrie proxy sous-jacente dans le shader. Une possibilité, plutôt que d'utiliser du branching, est de compiler des variantes différentes selon que l'on shade avec la lumière directionnelle ou des sources ponctuelles. Je vais modifier les constructeurs de *Shader* afin d'inclure un paramètre optionnel de sélection de variante, ce qui se traduira par un #define utilisable dans le shader lui-même. Le lighting shader sera réécrit pour abandonner le branching sur lt.u_light_type au profit d'une directive #ifdef. Et boom, c'est fait.

- La structure *ShaderResource* parse une string du type "lpass_exp.vert;lpass_exp.frag" et initialise des champs contenant les chemins d'accès complets vers les sources des shaders. La classe *Shader* utilise maintenant un unique constructeur avec comme argument un const ShaderResource&. Par ailleurs, un deuxième argument optionnel du constructeur de *ShaderResource* permet d'initialiser des flags qui seront inclus dans des directives #define dans la source du shader avant sa compilation, à l'instar de ce que fait le système de global defines. Le *LightingRenderer* définit maintenant deux variantes du même shader pour l'illumination :

```
lpass_dir_shader_(ShaderResource("lpass_exp.vert;lpass_exp.frag",
    "VARIANT_DIRECTIONAL")),
lpass_point_shader_(ShaderResource("lpass_exp.vert;lpass_exp.frag",
    "VARIANT_POINT")),
```

Ces deux variantes sont appelées séparément pour réaliser l'éclairage ponctuel et directionnel.

Plusieurs flags peuvent être définis, séparés par des point-virgules.

[06-11-18]

[BUG][fixed] Flickering Black Rectangle

Lors du dev de la reconstruction de position, j'observais un bug étrange, où sous certaines conditions d'orientation de la caméra un rectangle noir pouvait apparaître dans le ciel uniquement, et disparaissait sous la géométrie. Jess m'a aidé à reproduire le bug, et en trifouillant s'est rendu compte que désactiver l'effet bloom ne supprimait pas le bug comme je le pensais initialement, mais laissait un unique pixel noir à l'écran. L'effet bloom ne faisait qu'étaler un état indéfini ou singulier sur plusieurs pixels par convolution. En m'orientant de sorte à mettre le pixel noir au milieu de l'écran, j'ai remarqué que j'étais parfaitement aligné avec la direction de la lumère directionnelle. Le bug était donc vraissemblablement dû à un vecteur qui s'annule sous cette condition. Immédiatement le half-way vector utilisé dans le modèle Cook-Torrance m'a semblé être le suspect idéal :

```
    vec3 halfwayDir = normalize(viewDir + lightDir);
```

Clairement, ce vecteur va s'annuler lorsque lightDir = - viewDir. Ceci a pour effet de réduire le terme de Trowbridge-Reitz au carré de la rugosité, qui peut être nulle dans le ciel. Le fix consiste à seuiller le terme TrowbridgeReitzGGX :

```
1     return max(num / denom, 0.001);
```

[BUG][fixed] Anti-reflections

A une certaine heure de la nuit, on pouvait voir apparaître des réflections sombres aux tons bleus-violacés. Ceci était dû à l'interpolation de la brightness de la lumière directionnelle qui pouvait devenir négative. Cette valeur est maintenant seuillée dans le *DaylightSystem*.

[GUI]

Nouveau panneau de contrôle pour le *DaylightSystem*. On peut modifier l'heure de la journée, les paramètres orbitaux du Soleil et de la Lune, et les paramètres de la lumière directionnelle.

[07-11-18]

Git

J'ai initialisé un git. La procédure est la suivante :

1) Créer un compte github et un repository (https://github.com/ndoxx/wcore)
2) Cloner le git dans un dossier temporaire

```
1 >> cd tmp
  >> git clone https://github.com/ndoxx/wcore
```

3) Copier le .git dans le dossier wcore (ainsi que le README etc.)

```
>> cp tmp/.git WCore/
```

4) Créer un fichier .gitignore à la racine du projet et le remplir des dossiers et fichiers qu'on ne veut pas uploader.
5) Ajouter les sources

```
1 >> git add *
```

6) Vérifier le statut de ce qui va être commit

```
1 >> git status
```

7) Commit

```
1 >> git commit -m "first commit"
```

8) (Optionnel) Conserver le mot de passe (à faire seulement la première fois) :

```
1 >> git config credential.helper store
```

9) Push

```
1 >> git push origin master
```

Si l'étape 8 est effectuée, git ne demandera l'authentification que la première fois.

Sphères et domes texturés

Comme j'ai trouvé casse-couilles de deviner les coordonnées UV qui vont bien sur mes sphères procédurales, je me suis lancé une session Blender pour générer une sphère et un dome avec du UV mapping. Au bout de quelques heures de galère avec l'UV unwrapping (détaillé dans le cahier) j'ai fini par converger. Mon principal problème pour texturer une sphère a été le suivant : J'avais décidé de générer 2 îles pour l'UV unwrapping (une pour chaque hémisphère), localisées dans les deux moitiés respectives d'une texture. Une île est basiquement une moitiée de sphère applatie dans le plan. Les cercles extérieurs des 2 îles sont identifiés. Le problème avec cette approche est qu'il y a un saut de coordonnées UV au niveau des vertices de l'équateur, ce qui implique des erreurs d'interpolation et d'affreuses distortions des textures près de l'équateur in-game. Ma solution a été de "rip" la sphère pour forcer la duplication des vertices le long de l'équateur, chaque composante connexe est ensuite UV unwrap sur le même data block. Et là tout fonctionne. J'ai aussi généré un dôme texturé.

La texture elle-même est générée à partir d'un panorama (alors faut pas s'imaginer un joli paysage, ce sont mes gros doigts de codeur qui ont fait le boulot) déformé en coordonnées polaires sous Gimp et centré sur l'île.

Je me servirai du dôme pour rendre un **Sky dome** un peu plus tard. Je préfère cette approche à la skybox parce que je compte animer le ciel (à terme *DaylightSystem* servira à ça).

[12-11-18] Refactoring du système d'assets

[x] Parser un fichier XML pour localiser les assets [x] Remplacer le système de Texture::ASSET_MAP_ [ ] Gérer plusieurs définitions [x] _Material_ définit des grandeurs uniformes qui peuvent remplacer une texture unit, systématiser ceci afin de rendre les textures optionnelles. [x] Ce qui permettra de se débarrasser des textures par défaut. -> Toutes les textures PBR sont optionnelles. Si un *Material* ne possède pas l'une d'entre elles, un uniform est envoyé à la place.

Voici comment on déclare un asset dans assets.xml :

```
<Material>
    <Name>plop</Name>
    <Texture>
        <Albedo>plop_albedo.png</Albedo>
        <AO>plop_ao.png</AO>
        <Depth>plop_depth.png</Depth>
        <Metallic>plop_metal.png</Metallic>
        <Normal>plop_norm.png</Normal>
    </Texture>
    <Uniform>
        <Roughness>0.3</Roughness>
        <ParallaxHeightScale>0.15</ParallaxHeightScale>
    </Uniform>
    <Override>
        <NormalMap>true</NormalMap>
        <ParallaxMap>true</ParallaxMap>
    </Override>
</Material>
```

Essentiellement, il faut donner un nom au material (qui sera hashé et servira de resource ID, donc ce nom *doit* être unique), puis définir la source des données pour chaque grandeur de la pipeline PBR. La source peut être une image, alors on doit donner le nom de cette image dans le noeud *Texture*, ou bien une grandeur uniforme, auquel cas une valeur peut être précisée dans le noeud *Uniform*. D'autres options de shading sont transmissibles via le noeud *Uniform*, comme *ParallaxHeightScale* ou *Transperency*. Enfin, le normal mapping et le parallax mapping peuvent être désactivés via le noeud *Override*.

Les structures du fichier assets.xml sont parsées par *MaterialFactory*. Pour chaque material, un descripteur (struct *MaterialDescriptor*) est sauvegardé dans une hashmap, en association avec le resource id du material (hash du nom).

Un descripteur contient toutes les données nécessaires à la construction online d'un *Material*. *MaterialDescriptor* est déclaré dans material_common.h. Une sous-structure de *MaterialDescriptor* est *TextureDescriptor* qui stock les données relatives au texturing uniquement. En particulier, j'ai prévu une structure *TextureParameters* qui fixe les options OpenGL (filter, format, clamp…).

[TODO][x] Le jeu DOIT être exécuté depuis le dossier build sinon ça ne fonctionne pas. C'est dû aux nombreux paths hardcodés. Corriger ça.

[x] Réparer la SSAO sous **EXPERIMENTAL_POS_RECONSTRUCTION**. -> Par ailleurs, j'en ai eu marre de la lenteur de cet algo supposé rapide alors j'ai investigué. Si dans le shader SSAO.frag on remplace dans la boucle :

```
    //vec2 coord1 = reflect(SAMPLES[jj],randomVec)*rad;
2   vec2 coord1 = SAMPLES[jj]*rad;
```

En éliminant le reflect, donc, eh bien c'est foutrement plus rapide (bien que ça introduise des artéfacts visibles de près). Ceci est dû (vérifié) à la ligne

```
    vec2 randomVec = normalize(texture(noiseTex, texCoord*rd.v2_noiseScale).xy);
```

un peu plus haut qui se trouve optimisée à la compilation. En particulier, la multiplication par rd.v2_noiseScale. Quelques tests me montrent que plus rd.v2_noiseScale est grand ((15, 8.4375) pour mon ratio) et plus la SSAO prend du temps. Malheureusement, diminuer cette valeur supprime son utilité.

La texture bruit tessèle l'écran, et le noiseScale est l'échelle de tesselation. Plus il est grand, et plus on va chercher des texels loins les uns des autres. Le problème semble lié à l'accès texture random (voir [1]) qui diminie la cohérence spatiale et augmente les cache misses (si on ne sample que des texels voisins on maximise le cache use). Le problème peut être circonscrit en diminuant le rayon de la SSAO. En effet, je reviens à des temps honnêtes pour un rayon de 0.25 (plus bas et on a du banding).

Sources:

```
1 [1] https://stackoverflow.com/
  questions/38953632/slow-texture-fetch-in-fragment-shader-using-vulkan
```

[15-11-18] Better terrain

Le système de terrain sera amené à être complexifié par la suite (Delaunay triangulation + curvature based importance sampling, progressive mesh si nécessaire). Pour ça, je pense qu'il vaut mieux porter la génération de terrain dans une passe offline séparée, comme ça, les terrains feront partie intégrante de la content pipeline (possibilité de modifier sous blender…). Je pourrai donc réserver un gros système pour générer les différents chunks, appliquer des modifiers (érosion…) et faire le stitching en avance, avant d'exporter des mesh au format obj (et flat buffer plus tard) qui seront importées par le jeu. On pourrait imaginer un éditeur graphique standalone pour le terrain, capable d'exporter les mesh rapidement, de sorte que l'éditeur de jeu qui tourne séparément (genre sur mon deuxième écran que j'aurai un jour) puisse simplement recharger la map afin de visualiser rapidement les changements. -> Chaque chunk possèderait sa splat map et on pourrait la dessiner depuis l'éditeur de terrain. -> On pourrait aussi modifier la géométrie avec différents outils.

[21-11-18] Après le repos, la guerre.

L'application "wcore" (qui sera bientôt renommée) est maintenant une application hôte. Son main.cpp est localisé dans le dossier hosts à la racine. Les applications hôtes ont accès aux includes de WCore, et compilent pour l'instant les sources de WCore séparément. Quand j'aurai une API elles n'auront plus qu'à link une lib statique. Cette architecture de projet me permet de séparer maintenant les sources "engine" des sources applicatives.

Une nouvelle application hôte “ecs” vient de voir le jour. A l’image de “wcore” qui permet de tester le rendu et l’update d’une map simple avec de la géométrie essentiellement statique en s’appuyant sur de gros systèmes de WCore, “ecs” illustrera l’utilisation d’autres systèmes principaux tels que le système d’entités et de composants, en couplage fort avec un système de scripting qui reste à développer. La forme finale de l’application (graphique ou non) n’est pas encore définie à ce jour.

Les entités “drawable” seront rendues séparément dans la passe géométrique, au dessus de la géométrie statique. Tout ne sera pas entité dans le jeu, en particulier la géométrie statique non destructible doit pouvoir se passer de l’overhead. Donc il est raisonnable de penser que l’introduction d’entités dessinables se fera simplement au prix de quelques ajouts dans les classes *GeometryRenderer*, *Scene* et pourquoi pas *Chunk*, et ne nécessitera pas de refactor en profondeur. Toute la gestion des entités doit donc pouvoir s’envisager orthogonalement au rendu, ce qui est fort heureux.

[23-11-18] Better logger

Je vais implémenter un système de canaux de communication pour le logger, ce qui permettra de filtrer dynamiquement les messages de debug. Chaque instruction DLOGx pourra préciser un canal en argument et l’affichage console sera modulé en fonction des canaux actifs. L’UI du logger sera étendue pour proposer des cases à cocher pour chaque canal. Le système *Config* établira quels sont les canaux actifs au lancement. Chaque canal sera référencé par un hash string. De plus, j’imagine y joindre un système de verbosité (une valeur à 4 niveaux pour chaque canal) ce qui permettra de grouper des comportements tels que **DEBUG_TEXTURE** et **DEBUG_TEXTURE_VERBOSE** sous un même canal. Du coup, peut être que des sliders colleraient mieux dans l’UI…

```
       severity          critical    warning     low    detail
    verobsity level
           0                X
           1                X          X
           2                X          X          X
           3                X          X          X          X

                           DLOG[E,F]  DLOGW           DLOGx

[x] __DEBUG_TEXTURE__
[x] __DEBUG_TEXTURE_VERBOSE__
[x] __DEBUG_MATERIAL_VERBOSE__
[x] __DEBUG_MODEL__
[x] __DEBUG_MODEL_VERBOSE__
[x] __DEBUG_SHADER__
[x] __DEBUG_SHADER_VERBOSE__
[x] __DEBUG_TEXT__
[x] __DEBUG_KB__
[x] __DEBUG_SPLINES__
[x] __DEBUG_BUFFERS__
[x] __DEBUG_CHUNKS__
```

[26-11-18] L’important c’est de coder

Better logger cont’d

Donc les implémentations ont été réalisées sans souci. Le *Logger* peut enregistrer des canaux (struct *LogChannel*) référencés par des hashstr_t de leurs noms. Chaque canal comporte un nom en toutes lettres, un niveau de verbosité courant et un style d’affichage (pour la console et pour le debug GUI). Toutes les macros DLOGx ont été modifiées pour prendre en arguments supplémentaires un nom de canal et un indice de sévérité (enum *Severity*) allant de 0 à 3 (DETail, LOW, WARNing, CRITical). Un niveau de verbosité requis est calculé (verbosity_req = 3 - severity).

Si ce niveau est supérieur ou égal au niveau de verbosité courant pour le canal concerné, alors le message est affiché dans la console. Sinon il reste toutefois empilé pour l'écriture du fichier log. Le niveau de verbosité de chaque canal est modifiable via un menu contractible du widget 'Logging', par l'intermédiaire de sliders générés à la volée. Comme le montre le tableau précédent, les messages critiques continuent de s'afficher quel que soit le niveau de verbosité. Ainsi on peut considérer qu'un canal est désactivé quand sa verbosité est à 0, pour autant, on ne masque pas les messages ayant une sémantique d'erreur ou d'erreur fatale.

Le fichier config.xml définit de nouveau noeuds *root.debug.channel_verbosity.[channel_name]* dont la valeur permet de fixer la verbosité initiale de chaque canal.

Le premier canal enregistré par le système est le canal "core" qui possède une verbosité initiale maximale. Ce canal un peut spécial n'est pas configurable via config.xml. Il permet au logger lui-même et au parser XML de pouvoir lancer des messages debug avant l'enregistrement des autres canaux qui a lieu après le parsing de config.xml, lors de l'initialisation de *Config*.

A l'enregistrement d'un canal, une couleur aléatoire est générée qui lui servira de style pour l'affichage console et sa représentation dans l'UI (oui, je me suis fait chier). La couleur est générée dans l'espace HSL depuis le hash name du canal utilisé comme seed et convertie en RGB sous des formats utilisables par la console (ANSI string) et l'UI (float array). -> voir colors.h/cpp pour les nouvelles fonctions.

Des macros supplémentaires DLOGS et DLOGES permettent de définir une section avec un titre et une fin de section avec des '——'. Eviter d'utiliser des tags dans le titre, ça perturbe le formatage de la fin de section. Les sections sont là pour rendre la sortie debug plus lisible, mais ne hiérarchisent pas l'information de debug (celà peut être amené à changer).

L'affichage console est beaucoup plus clair :

```
[timestamp][channel][icon][Message]

[0.000208][cor]     [Config] Beginning configuration step.
[0.000252][cor]        Self path: /home/ndx/dev/WCore/bin/wcore
[0.000285][cor]        Root path: /home/ndx/dev/WCore
[0.000311][cor]        Config path: /home/ndx/dev/WCore/config
[0.000329][cor]     [Config] Parsing xml configuration file.
[0.000341][cor]     [XML] Parsing xml file:
[0.000355][cor]        /home/ndx/dev/WCore/config/config.xml
[0.000566][cor]     ------------------------------------
[0.154930][cor]     [XML] Parsing xml file:
[0.154954][cor]        /home/ndx/dev/WCore/res/levels/assets.xml
[0.181065][inp]     [InputHandler] Parsing key bindings.
[0.181103][cor]     [XML] Parsing xml file:
[0.181109][cor]        /home/ndx/dev/WCore/config/keybindings.xml
[0.181163][inp]        Parsing category: DebugControls
[0.181183][inp]     [PRESS] LEFT_SHIFT -> Freecam: Faster camera movements.
[0.181203][inp]     [RELEASE] LEFT_SHIFT -> Freecam: Slower camera movements.
```

Les canaux sont affichés en abrégé (3 premières lettres) avec une couleur de fond correspondant à leur styles, l'icône est maintenant un caractère unicode simple, chaque ligne commence par un timestamp en vert, les sections sont en noir sur fond blanc pour un bon contraste…

Les defines **DEBUG_x** ont tous été supprimés au profit de **DEBUG**, les tags **PROFILING_x** n'ont pas été touchés.

Il n'est plus nécessaire de recompiler tout le putain de code pour filtrer l'info de debug.

[filesystem] Cauchemar cannabinique

De longues heures à pester contre le mauvais support de filesystem par Clang et la lenteur extraordinaire d'adoption des nouveaux features du C++ par les vendors, parce que trop à l'ouest pour me rendre compte de mes propres

conneries à cause d'une mauvaise lecture initiale des messages d'erreur… C'est vraiment ça le sujet. Passé ce stade, le refactor fut aisé.

Tous les systèmes utilisent des fs::path en lieu et place des strings pour faire référence à un chemin d'accès. Certains noms de fichiers peuvent encore être passés en const char* mais le chemin complet est en fs::path jusqu'au passage à un objet fstream.

Le système *Config* gère une map de chemins d'accès en fs::path. Chaque path est vérifié à l'initialisation, de sorte que tous les chemins d'accès pointés par *Config* sont valides. Des dossiers peuvent être déclarés dans config.xml dans les noeuds *root.folders.[name]*, et peuvent être accédés in-game via :

```
    fs::path file_path = io::get_file(HS_("root.folders.[name]"), filename);
```

Le fichier io_utils.h déclare en effet une fonction io::get_file() qui prend en argument un config node en hashstr_t et un nom de fichier en const char* et retourne un fs::path vers le fichier (ou un chemin vide si erreur).

Ce système d'accès centralisé devra être augmenté pour aller chercher des fichiers contenus dans une archive. Les accès fstream devront donc être repliés derrière les fonctionnalités de io_utils.h. [x] On pourra imaginer des fonctions de la même allure que io::get_file() qui se chargent de retourner des buffers (texte ou binaire) plutôt que des chemins d'accès. -> Manque plus qu'à faire fonctionner ça avec *PngLoader*, libpng me fait des misères (IHDR CRC Error).

Self-path bootstrap

La méthode Config::init() est la permière méthode appelée dans une application :

```
1 int main(int argc, char const *argv[])
  {
3     // Parse config file
      CONFIG.init();
5     // ...
```

Cette méthode va dans un premier temps chercher à localiser le fichier exécutable en absolu grâce à un appel système (OS-dependant). Sous Linux on utilise la fonction readlink() de unistd.h :

```
1     char buff[PATH_MAX];
      std::size_t len = ::readlink("/proc/self/exe", buff, sizeof(buff)-1);
3     if (len != -1)
      {
5         buff[len] = '\0';
          return fs::path(buff);
7     }
      else
9     {
          DLOGE("Cannot read self path using readlink.", "core", Severity::CRIT);
11        return fs::path();
      }
```

Comme le chemin retourné comprend le nom de fichier de l'exécutable, le dossier racine est déduit comme le parent du parent de ce path. La méthode init() vérifie ensuite l'existence d'un dossier 'config' à la racine, duquel le fichier config.xml pourra être lu, après quoi tous les chemins d'accès sont connaissables pour le reste du système. C'est cette même méthode qui procède à l'initialisation des canaux du *Logger*.

START_LEVEL

Une nouvelle variable globale GLB.START_LEVEL est initialisée via la fonction d'argument parsing de 'rd_test' (argument.h/cpp maintenant dans le dossier host), ce qui permet de choisir le niveau à charger depuis la ligne de commande :

```
>> ../bin/wcore -l crystal
>> ../bin/wcore -l tree
```

Le nom des fichier de niveau suit la syntaxe

```
l_[name].xml
```

La première ligne cherchera donc à charger le fichier l_crystal.xml et la deuxième le fichier l_tree.xml. Le dossier contenant les niveaux est accessible depuis le noeud de config *root.folders.level*.

[27-11-18]

Exceptions

Le programme est exception-free. J'utilisais les exceptions sans jamais les intercepter pour mettre fin au programme en cas d'erreur fatale. C'est un peu inutile, le fichier error.h définit une fonction fatal() pour sortir proprement du programme en affichant un message si nécessaire. A noter que le message n'est pas transmis au *Logger*, il convient d'appeler DLOGF avant fatal() pour logger un message d'erreur.

[28-11-18] API culture

Messaging

Tout mon code utilise le messaging system pour réagir aux événements clavier / souris. *GLContext* a été scindé en 2 parties : *GameLoop* dans engine_core.h/cpp qui définit la boucle de jeu, et *Context* dans context.h/cpp qui fait l'interface avec GLFW. *InputHandler* est maintenant un membre de *GameLoop*, et produit des événements wcore quand la souris est bougée/cliquée ou bien qu'une touche clavier a été pressée, pour laquelle il existe un binding.

Un événements clavier ne comporte comme information que le nom du binding :

```
struct KbdData : public WData
{
    KbdData(hash_t keyBinding): key_binding(keyBinding) {}
    hash_t key_binding;
};
```

Un événements souris contient le déplacement depuis la dernière frame et un bitset pour l'état des boutons :

```
struct KbdData : public WData
{
    // ctor
    // to_string()
    float dx = 0.f;
    float dy = 0.f;
    std::bitset<4> button_pressed;
};
```

Macro utilisée pour remplacer les instructions register_action par un bloc “case xxx:” dans un switch :

```
handler.register_action\((.+), \[&\]\(\)\s+\{\s+(.+)\s+\}\);
case $1:\n\t\t$2\n\t\tbreak;
```

De fait, l'ensemble de la fonction main() a pu être globalement réduit et j'ai pu commencer à travailler sur l'API du moteur.

[API] Ayéééé !

Le fichier wcore.h définit l'API de la libwcore utilisable par mes applications hôtes. La classe *Engine* définit pour l'instant 3 méthodes pour initialiser le moteur (Init()), charger le premier niveau (LoadStart()) et lancer la game loop (Run()). J'utilise l'idiome Pimpl pour masquer l'implémentation derrière un pointeur opaque :

```
// wcore.h
class WAPI Engine
{
    // ...
private:
    struct EngineImpl;
    std::unique_ptr<EngineImpl> eimpl_; // opaque pointer
};

// wcore.cpp
struct Engine::EngineImpl
{
    // ctor, dtor, init()
    GameLoop*       game_loop;
    SceneLoader*    scene_loader;
    RenderPipeline* pipeline;
    DaylightSystem* daylight;
    ChunkManager*   chunk_manager;
}

Engine::Engine():
eimpl_(new EngineImpl)
{

}
```

La classe *Engine* de l'API déclare une structure interne *EngineImpl* qui est définie dans l'implémentation, et un pointeur vers cette structure. La structure est allouée à la construction de *Engine* et lazy-initialized après le parsing des arguments du programme.

Une fonction externe GlobalsSet() permet de muter les variables de la classe *Globals* (utile pour que le parser de arguments.cpp puisse faire son boulot).

```
// wcore.h
extern "C" void WAPI GlobalsSet(hashstr_t name, const void* data);

// wcore.cpp
void GlobalsSet(hashstr_t name, const void* data)
{
    switch(name)
    {
        default:
            warn_global_not_found(name);
            break;
        case HS_("SCR_W"):
            GLB.SCR_W = *reinterpret_cast<const uint32_t*>(data);
            break;
        case HS_("SCR_H"):
            GLB.SCR_H = *reinterpret_cast<const uint32_t*>(data);
            break;
        case HS_("SCR_FULL"):
            GLB.SCR_FULL = *reinterpret_cast<const bool*>(data);
```

```
                break;
            case HS_("START_LEVEL"):
                char* value = const_cast<char*>(reinterpret_cast<const
                    char*>(data));
                GLB.START_LEVEL = value;
                break;
        }
    }
```

Le main() de sandbox se réduit à :

```
#include "wcore.h"
#include "arguments.h"

int main(int argc, char const *argv[])
{
    wcore::Engine engine;
    engine.Init(argc, argv, sandbox::parse_program_arguments);
    engine.LoadStart();
    return engine.Run();
}
```

Le 3ème argument de wcore::Init est un pointeur sur fonction vers un parser personnalisé (arguments.h/cpp). Cet argument est optionnel (default nullptr).

Je n'ai pas encore ressenti le besoin de déclarer mon main() dans la lib (**entry point**).

Les CMakeLists.txt ont dû être modifiés en profondeur pour définir une nouvelle cible en shared lib (target wcore) et une application "sandbox" qui link avec libwcore. *"sandbox" est basiquement l'ancienne application nommée "wcore", et "wcore" est maintenant le nom de la lib dynamique.* Donc maintenant on doit faire :

```
>> cd build
>> cmake ..
>> make wcore
>> make sandbox
```

make wcore va générer libwcore.so (et .so.1 et .so.1.0.1) dans le dossier lib.

La seule petite galère a été de faire fonctionner ça avec freetype qui n'avait pas été compilée en Position Independent Code (-fPIC). Il a fallu recompiler freetype avec cette option :

```
>> ./configure CXXFLAGS=-fPIC CFLAGS=-fPIC LDFLAGS=-fPIC CPPFLAGS=-fPIC
>> make
```

Freetype est gérée à la va-vite par le projet : les includes ont été copiés dans vendor/freetype depuis /usr/local/include/freetype et la libfreetype.a dans le dossier lib

[30-11-18] Pybind11

Je parviens à me servir de Pybind11 pour appeler des fonctions C++ depuis un contexte Python embarqué (voir ~ /practice/pybind11/embedded). Ca se complique quand je veux utiliser C++17 (voir ~ /practice/pybind11/embedded_cpp17).

Dans le CMakeLists.txt il faut dire à pybind11 d'utiliser le standard C++17 (à ce jour expérimental).

```
    set(PYBIND11_CPP_STANDARD -std=c++1z)
    add_subdirectory(pybind11)
```

Un simple programme refusera de compiler :

```
In file included from /home/ndx/practice/pybind11/embedded_cpp17/main.cpp:1:
In file included from
   /home/ndx/practice/pybind11/embedded_cpp17/pybind11/include/pybind11/embed.h:12:
/home/ndx/practice/pybind11/embedded_cpp17/pybind11/include
/pybind11/pybind11.h:1000:9: error:
      no matching function for call to 'operator delete'
        ::operator delete(p, s, std::align_val_t(a));
```

Le problème semble connu (voir [1]) mais depuis moins d'un mois, et aucune solution n'est apportée par l'auteur de pybind11 (Wenzel Jakob).

Quand on va voir dans pybind11.h vers la ligne 1000 on voit ceci :

```
inline void call_operator_delete(void *p, size_t s, size_t a) {
    (void)s; (void)a;
#if defined(PYBIND11_CPP17)
    if (a > __STDCPP_DEFAULT_NEW_ALIGNMENT__)
        ::operator delete(p, s, std::align_val_t(a));
    else
        ::operator delete(p, s);
#else
    ::operator delete(p);
#endif
}
```

Effectivement, quand C++17 est définit comme le standard à utiliser, delete est appelé avec 2 ou 3 arguments supplémentaires (taille et alignement), sûrement pour coller aux nouvelles signatures de delete introduites par le standard(voir [2]) :

```
    void operator delete(void*, std::size_t, std::align_val_t);
    void operator delete[](void*, std::size_t, std::align_val_t);
```

On repère ces signatures dans le header C++ /usr/include/c++/8/new à partir de la ligne 159 on a :

```
#if __cpp_sized_deallocation
void operator delete(void*, std::size_t, std::align_val_t)
  _GLIBCXX_USE_NOEXCEPT __attribute__((__externally_visible__));
void operator delete[](void*, std::size_t, std::align_val_t)
  _GLIBCXX_USE_NOEXCEPT __attribute__((__externally_visible__));
#endif // __cpp_sized_deallocation
```

Donc il y a fort à parier que dans mon cas, la macro **cpp_sized_deallocation** n'est pas définie. Cette macro est décrite dans [4], et semble être un feature du C++14… Je ne sais pas ce que je dois comprendre, il me manquerait un feature de C++14 ?

Modifier le code de pybind11.h comme suit :

```
inline void call_operator_delete(void *p, size_t s, size_t a) {
    (void)s; (void)a;
    ::operator delete(p);
}
```

supprime l'erreur, mais je ne fais aucune confiance à ce fix.

Solution

AHA ! Clang ne supporte pas le feature "sized deallocation" par défaut. Il faut ajouter le flag -fsized-deallocation pour que ce feature soit activé (et **cpp_sized_deallocation** définit). Donc la ligne suivante est requise dans le CMakeLists.txt :

```
    add_definitions(-fsized-deallocation)
```

J'ai posté mon fix dans [1], on verra bien ce que les gens en disent. -> Ils confirment que mon fix est le bon.

Script directory

Par défaut, pybind inclut le "working directory" dans le sys.path, ce qui permet à un script localisé à l'endroit de l'exécution du programme d'intéragir avec celui-ci. Or je souhaite plutôt localiser tous mes scripts dans un dossier spécifique. Il me faut donc inclure le chemin d'accès du dossier scripts dans le sys.path au lancement de l'application. Ma méthode un peu dégueu sur les bords consiste à faire :

```
    fs::path self_path = get_selfpath();
    fs::path base_path = self_path.parent_path().parent_path();
    fs::path scripts_path = base_path / "scripts/";
    assert(fs::exists(scripts_path));

    std::cout << "Scripts directory: " << scripts_path.string() << std::endl;

    std::stringstream ss;
    ss << "import sys" << std::endl;
    ss << "sys.path.insert(0, \"" << scripts_path.string() << "\")";
    py::exec(ss.str());

    std::cout << "Updated Python path: " << std::endl;
    py::module sys = py::module::import("sys");
    py::print(sys.attr("path"));
```

Avec la même fonction get_selfpath() que dans config.cpp (appel à ::readlink()). Ca permet de forger l'instruction suivante en Python :

```
import sys
sys.path.insert(0, "path/to/scripts/directory")
```

Alors la ligne :

```
py::print(sys.attr("path"));
```

affiche le path mis à jour avec succès :

```
['/home/ndx/practice/pybind11/embedded_cpp17/scripts/', '/usr/lib/python35.zip',
    '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-linux-gnu',
    '/usr/lib/python3.5/lib-dynload',
    '/home/ndx/.local/lib/python3.5/site-packages',
    '/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages', '.']
```

Et les scripts localisés dans le dossier scripts sont bien dans le path.

Sources :


```
[1] https://github.com/pybind/pybind11/issues/1604
[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0035r4.html
[3] https://github.com/pybind/pybind11/issues/948
[4] https://en.cppreference.com/w/cpp/feature_test
```

[01-12-18]

ECS API

Je me retourne les méninges à essayer d'imaginer une API pour l'ECS. Le souci est que je repose énormément sur des templates, ce qui m'oblige à exposer une partie de la business logic dans l'interface. Je vais essayer de lister ce qui mérite d'être exposé (noté X), et ce qu'on est alors forcé d'exposer en l'état à cause d'une dépendance de l'interface (noté F), ça m'aidera à y voir plus clair.

```
                              E    Raison
* wcomponent.h
    class WComponent          X    Pour qu'un composant user puisse en hériter.
    T* create<T>()            F    Dépendance dans l'interface de WEntity
    void destroy(cmp* )       F    --
    REGISTER_COMPONENT        X    Pour pouvoir enregistrer les composants user.
* basic_components.h
    WCTransform               X    Pour pouvoir utiliser ce type côté user.
    Transformation            F    Par héritage de WCTransform (aïe ?)
* component_detail.h
    getComponentRegistry()    F    Dépendance dans la factory de
    cmp* createComponent<T>() F    l'interface wcomponent.h
    struct RegistryEntry<T>   F    --
* wentity.h
    WEntity::
        add_component<T>()    X    Pour pouvoir intéragir avec les composants
        get_component<T>()    X    d'une entité, en connaissance des types.
        has_component<T>()    X    --
```

Que les factories soient exposées, passe encore. L'ensemble de component_detail.h j'ai déjà un peu plus mal au cul. Mais *Transformation* c'est un autre genre d'emmerdes. Basiquement, si on expose cette classe, alors on doit exposer toutes les classes de maths. Et si la situation se reproduit avec un autre composant qui hérite d'une classe de WCore ça va être l'explosion de caca. -> On peut imaginer une relation de composition entre *WCTransform* et *Transformation*. Un wrapper et du PImpl, emballez c'est pesé.

[09-12-18] Cotire

J'utilise le module CMake Cotire (compile time reducer) pour accélérer le build en automatisant la génération d'un precompiled header et de targets unity builds.

Un dossier "cmake" est présent à la racine, qui contiendra tous les modules CMake customs que je vais intégrer au projet. Le CMakeLists.txt à la racine doit donc comporter les deux lignes suivantes :

```
    set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake/")
    include(cotire)
```

La première initialise le path parcouru en premier par CMake pour aller chercher un module lors d'un appel à include() ou find_package(). La seconde va exécuter cotire.cmake localisé dans le dossier de modules "cmake".

Pour accélérer un build, il suffit de rajouter :

```
    cotire([target_name])
```

juste après le target_link_libraries(), genre cotire(ecs), cotire(sandbox), cotire(wcore)…

Et la suite c'est de la putain de magie noire. Le script va de ce que j'ai compris parcourir les sources et rechercher les headers externes (qui ne seront a priori jamais modifiés comme , et méritent donc d'être dans un PCH afin d'éviter leur parsing à chaque putain d'include), et générer un PCH automatiquement sur lequel la cible sera construite.

De plus, Cotire génère pour chaque target une target unity build avec la même sortie et les mêmes paramètres que la target de base, mais qui compile méga plus vite. Pour construire depuis un build unity il suffit de lancer :

```
>> make target_name_unity
```

exemples :

```
>> make wcore_unity
>> make sandbox_unity
```

Le build est monolithique (un seul fichier cpp sera compilé, le fichier unity). On ne peut plus suivre la progression et le compilo mouline longtemps sans rien afficher ce qui est déconcertant, mais la compilation est beaucoup plus rapide.

[12-12-18] Post-processing

ANNIV DE JESS DEMAIN

J'ai implémenté quelques goodies (low hanging fruits) dans le *PostProcessingRenderer* : * Effet vignette -> Consiste à assombrir et désaturer les bords et coins de l'écran pour émuler cet artéfact bien connu des caméras. Mon étape de saturation a été regroupée à cet endroit du shader :

```
    float vignette =
       mix(pow(16.0*texCoord.x*texCoord.y*(1.0-texCoord.x)*(1.0-texCoord.y),
       rd.f_vignette_falloff), 1.0f, rd.f_vignette_bal);
    out_color = saturate(out_color, max(0.0f,rd.f_saturation + vignette - 1.0f));
    out_color *= vignette;
```

- Contraste -> Transformation de dynamique. J'ai choisi un modèle linéaire rapide et simple :

```
out_color = ((out_color - 0.5f) * max(rd.f_contrast, 0)) + 0.5f;
```

- Vibrance -> Sorte de saturation intelligente qui ne sursature pas les couleurs déjà saturées :

```
vec3 vibrance_rgb(vec3 color_in)
{
vec3 color = color_in;
float luma = dot(W, color);

float max_color = max(color.r, max(color.g, color.b)); // Find the strongest color
float min_color = min(color.r, min(color.g, color.b)); // Find the weakest color

float color_saturation = max_color - min_color; // The difference between the two
    is the saturation

// Extrapolate between luma and original by 1 + (1-saturation) - current
vec3 coeffVibrance = vec3(rd.v3_vibrance_bal * rd.f_vibrance);
color = mix(vec3(luma), color, 1.0 + (coeffVibrance * (1.0 - (sign(coeffVibrance)
    * color_saturation))));

return color;
}
```

  Un paramètre de force et un paramètre de balance pour chaque canal de couleur est disponible. Les balances servent à pomper légèrement un canal donné.

- Aberration chromatique -> Emule cet artéfact dû au stigmatisme approximatif des optiques de caméras :

```
vec3 chromatic_aberration(vec3 color_in, vec2 texcoord)
{
vec3 color;
// Sample the color components
color.r = texture(screenTex, texcoord + (rd.f_ca_shift / rd.v2_frameBufSize)).r;
color.g = color_in.g;
color.b = texture(screenTex, texcoord - (rd.f_ca_shift / rd.v2_frameBufSize)).b;

// Adjust the strength of the effect
return mix(color_in, color, rd.f_ca_strength);
}
```

Essentiellement, on dédouble l'image avec un bord rouge à droite et bleu à gauche. Le shift et l'intensité de l'effet sont paramétrables. Cet effet est appliqué juste avant bloom.

Tous ces effets sont configurables en temps réel dans l'UI.

[PingPongBuffer]

J'ai enfin une nouvelle classe *PingPongBuffer* capable depuis un *BufferModule* en entrée/sortie et une politique d'update du shader sousjacent, d'exécuter un nombre arbitraire de passes aller et retour (ping pong). C'est comme ça qu'on peut implémenter une approx de flou Gaussien en O(2N) avec des filtres séparables (plutôt qu'en O(N^2)). Une politique d'update est une classe qui définit une fonction update avec cette signature :

```
    void update(Shader& shader, bool pass_direction);
```

La classe *BlurPassPolicy* permet de contrôler un shader *blurpass* (y-compris sa variante **VARIANT_COMPRESS_R** utilisé par la SSAO).

Cette classe est utilisée dans la branche expérimentale de Variance Shadow Mapping pour remplacer l'implémentation ad-hoc précédente.

[SSAO] Ajout d'un paramètre de biais vectoriel de la distribution d'échantillons

Pour solutionner temporairement mon souci d'auto-occlusion (dont la cause est pour l'instant plus ou moins inconnue), j'ai ajouté un paramètre qui va plus ou moins tirer les échantillons en arrière de la normale. Ce faisant, les auto-occlusions sont les premières à disparaître. La SSAO est configurable en temps réel dans l'UI. Les paramètres sont les suivants : * Radius -> Rayon de recherche d'occludeurs (mis en perspective via une division par z). * Scalar bias -> Seuil du produit scalaire * Vector bias -> Paramètre de lerp pour pousser les échantillons dans le sens inverse de la normale, ainsi les échantillons d'auto-occlusion d'une même face seront annulés plus facilement par le seuil scalaire. * Intensity -> Intensité de l'effet. * Scale -> Contrôle le falloff de l'occlusion en fonction du carré de la distance. * Blur passes -> Contrôle le nombre d'itérations de flou, rendu possible grâce à la nouvelle classe *PingPongBuffer*. * Compression -> Contrôle de compression dynamique (gamma transform) sur l'occlusion pour contrebalancer la réduction d'intensité inhérente aux passes de flou.

[SSDO] Expérimentation avec l'occlusion directionnelle

J'ai modifié légèrement le shader de SSAO pour aller capturer la couleur des occludeurs et selon leur orientation, déterminer un taux de "bleeding" :

```
vec4 directional_occlusion(vec2 texCoord, vec2 offset, vec3 frag_pos, vec3
    frag_normal)
{
    vec3 diff = get_position(texCoord + offset) - frag_pos;
    diff = mix(diff, -frag_normal, rd.f_vbias);
    vec3 v = normalize(diff);
```

```
    float d2 = dot(diff,diff)*rd.f_scale;

    float occlusion_amount = max(0.0f,
        dot(frag_normal,v)-rd.f_bias)*(1.0f/(1.0f+d2));

    vec3 sample_normal = decompress_normal(texture(normalTex, texCoord +
        offset).xy);
    float bleeding = max(0.0f,dot(sample_normal, -frag_normal)); // Scale with
        distance
    vec3 first_bounce = bleeding * texture(albedoTex, texCoord + offset).rgb;
    return vec4(first_bounce, occlusion_amount);
}
```

J'ai abandonné l'idée assez vite car l'overhead est énorme chez-moi (je dois être fill-bound à cause des 3 canaux supplémentaires que cela requiert). De fait, l'implémentation est à l'arrache (même pas mis le bleeding à l'échelle en fonction de la distance, ni fait une intégration propre dans la pipeline PBR). Cependant j'ai pu observer un début d'approx de GI dans mon moteur et c'était rafraîchissant. On garde ça sous le coude au cas où.

[15-12-18]

Toolchain

Une toolchain alternative utilisant clang 6 (au lieu de 7) peut être utilisée par CMake via :

```
>> cmake -DCLANG6=1 ..
```

Les instructions relatives aux paths sont regroupées dans les fichiers toolchain_clang6.cmake et toolchain_clang7.cmake. Ces fichiers sont simplement inclus (c'est sûrement très mal) via

```
if(DEFINED CLANG6)
  include(toolchain_clang6)
else()
  include(toolchain_clang7)
endif()
```

J'ai essayé avec un SET(-DCMAKE_TOOLCHAIN_FILE /path/to/file) mais ça n'a rien donné. A retenter, c'est a priori comme ça qu'on fait (cross compilation). Mais les serveurs de chez CMake sont down, donc pas de docu pour moi pour l'instant :p

On the fly Gaussian kernels

L'include gaussian_blur.glsl définit une nouvelle fonction convolve_kernel_separable() qui permet une convolution avec un filtre séparable dont les coefficients du noyau (supposé symétrique) et le nombre de coefficients sont précisés en argument. Du coup blurpass.frag a subi un petit lifting :

```
struct GaussianKernel
{
    float f_weight[KERNEL_MAX_WEIGHTS];
    int i_half_size;
};
uniform GaussianKernel kernel;

void main()
{
    vec3 result = convolve_kernel_separable(kernel.f_weight, kernel.i_half_size,
                                            inputTex, texCoord,
```

```
                                                v2_texelSize, horizontal);
    // ...
}
```

Pour envoyer les uniforms, c'est dans BlurPassPolicy::update():

```
    shader.send_uniform<int>(H_("kernel.i_half_size"), kernel_.get_half_size());
    shader.send_uniform_array(H_("kernel.f_weight[0]"), kernel_.data(),
        kernel_.get_half_size());
```

BlurPassPolicy contient un nouveau membre kernel_ de type *GaussianKernel* (voir gaussian.h).

GaussianKernel implémente les moyens de calcul d'un noyau Gaussien depuis une taille de noyau et un écart-type sigma. Le calcul des coefficients du noyau se fait par intégration d'une distribution normale sur n intervalles avec n = (k+1)/2 et k la taille du noyau. Le noyau Gaussien étant symétrique de part et d'autre de x=0, on a en effet besoin que des coefficients des bins positifs. L'intégrale est approchée par la méthode de Simpson (voir mes notes dans le cahier), dans numeric.h/cpp: integrate_simpson() (optimisée).

Le noyau Gaussien et la fonction d'intégration sont unit testés dans catch_numeric.cpp -> target test_math.

Il s'ensuit que la taille du noyau Gaussien et le sigma sont aisément configurables dans le GUI section SSAO/blur.

[17-12-18]

- Shadow map:
  - Slope-scaled depth bias
  - Normal offset -> Beaucoup moins de Peter-Panning
  - GUI control
- Better Bloom
  - GUI control
- TODO :
  - UBO voir : https://github.com/TReed0803/QtOpenGL/blob/master/resources /shaders/ubo/Global-Buffer.ubo [x] Unproject click

[19-12-18]

Custom cursor

La nouvelle classe *GuiRenderer* se charge pour l'instant uniquement de l'affichage du curseur custom. Cet affichage a lieu après celui du GUI debug (plus tard, seul le curseur sera affiché après le GUI debug, mais d'autres éléments de *GuiRenderer* pourraient être affichés avant).

Le curseur est au stade expérimental. En mode windowed, je peux détecter la sortie du curseur de l'écran alors un event mouse focus est lancé. Alors *GameLoop* en réponse à l'event va rétablir l'affichage du curseur hardware. Le problème c'est que le curseur apparaît DANS la fenêtre à une position que je ne contrôle pas (genre dernière position de sortie d'écran), alors un autre event mouse focus est lancé, et *GameLoop* croit que le curseur est re-rentré dans la fenêtre. C'est très certainement dû au curseur virtuel de GLFW qui n'est pas synchronisé avec le curseur réel. Bref, c'est délicat à régler, d'autant plus que le comportement des systèmes relativement aux inputs est très largement event driven. C'est pour ça que le curseur custom est désactivé par défaut. L'option *root.gui.cursor.custom* de config.xml permet d'activer ce feature.

-> Il me semble avoir compris l'origine du léger décalage (non constant sur l'écran) que subissait le curseur en windowed full screen. Dans ce mode, la fenêtre ne mesure PAS la résolution demandée de 1920x1080 mais moins (1855x1056 chez-moi). Donc *tous* les systèmes qui utilisent GLB.SCR_W et GLB.SCR_H pour initialiser des tailles de buffer ou que sais-je, commettent une erreur. Un peu connement j'ai créé une nouvelle paire de globaux GLB.WIN_W et GLB.WIN_H pour contenir les bonnes tailles de fenêtre au lieu de simplement remplacer

GLB.SCR_W et GLB.SCR_H. Mais tous les systèmes utilisent les WIN_x et tout fonctionne. En particulier le décalage curseur réel / curseur custom disparaît. Sauf en mode windowed full screen où on a un offset constant sur tout le putain d'écran… Y a-t-il une taille codée en dur quelque part dans le code ? -> Non, c'est juste que quand j'écris les GLB.WIN_x dans *Context* juste après avoir ouvert la fenêtre, la taille que je récupère est celle que j'ai demandé, et non la taille réelle, car la fenêtre n'est pas encore ouverte et GLFW ne connaît pas encore la taille qui sera allouée par l'OS (noter que le windowed full screen est OS spécifique). J'ai donc choisi d'attendre 100ms avant de query la taille. Ce n'est **PAS SAFE**. -> Il faut trouver un moyen de faire ça avec un callback. -> Noter que GLB.WIN_x sont obtenus avec glfwGetFramebufferSize() et non glfwGetWindowSize() (il se trouve que chez-moi, les 2 coincident). Ces noms sont mauvais.

Ray Casting

Je dois mettre au point le mécanisme de sélection au curseur de l'éditeur. Il me faut pouvoir lancer un rayon (struct *Ray*) depuis des coordonnées écran (grâce à la classe *RayCaster*), tester la collision de ce rayon avec tous les AABB/OBB qui sont dans le frustum de la caméra, et retourner dans un premier temps une liste, dans un second temps un unique handle (à définir) vers un objet de la scène. [x] Il faut pouvoir mettre en cache les tests de collision Frustum/AABB/OBB produits par le *GeometryRenderer*. Peut être même qu'une passe externe préalable lors de la phase update est de mise. -> Scene::visibility_pass() fait ça et est appelée dans Scene::update(). Un flag visible_ de *Model* est alors modifié selon qu'il est ou non dans le view frustum. *GeometryRenderer* surveille pmodel->is_visible() afin de cull. [x] Dans un premier temps, il convient d'afficher le rayon qui vient d'être tiré. *DebugRenderer* doit pouvoir afficher ce type de primitives avec un TTL. -> Un objet peut être un *Model* statique, ou plus tard une *Entity* et le test peut échouer. Je pense automatiquement à un type retour std::optional> mais c'est un peu lourdingue syntaxiquement, faut bien l'avouer ! Un std::variant en utilisant int comme tag type pour signifier l'échec ? -> Mieux, y a un type std::monostate fait pour pouvoir default initialize un std::variant. Donc std::variant. Ex :

```
    // Without the monostate type this declaration will fail.
    // This is because S is not default-constructible.
    std::variant<std::monostate, S> var;
    // var.index() is now 0 - the first element
    // std::get<S> will throw! We need to assign a value
    var = 12;
    std::cout << std::get<S>(var).i << '\n';
```

-> 12

On pourrait imaginer le dispatching suivant pour les valeurs de retour :

```
struct Visitor
{
   void operator()(Model){}
   void operator()(Entity){}
   void operator()(std::monostate){}
};

std::variant<std::monostate, Model, Entity> ret = get_ray_cast_result();
std::visit(Visitor{}, ret); // invokes the int overload
std::visit(Visitor{}, ret); // ... and the double overload
std::visit(Visitor{}, ret); // ... and finally the std::monostate overload
```

[20-12-18]

Debug display requests: segments

Je veux pouvoir afficher des segments depuis le *DebugRenderer*. Il s'agira de tracer in-game les rayons produits par le ray casting.

Le *DebugRenderer* stocke maintenant une liste de *DebugDrawRequest* qui correspondent à des requêtes d'affichage de primitives mises en queue par d'autres systèmes. Des fonctions request_draw_x() permettront d'empiler un ordre d'affichage en spécifiant des informations géométriques, une couleur et une durée de vie de la primitive (TTL). En particulier :

```
void request_draw_segment(const math::vec3& world_start,
                          const math::vec3& world_end,
                          int ttl = 60,
                          const math::vec3& color = math::vec3(0,1,0));
```

Permettra de demander l'affichage d'une ligne spécifiée par les coordonnées world de ses extrémités. Afin de pouvoir ne stocker qu'un seul segment dans le VBO, il m'a fallu calculer une matrice affine qui transforme un segment unitaire (selon x) en un segment arbitraire spécifié par ses extrémités. Tout le calcul est spécifié dans le cahier et la fonction qui produit de telles matrices a été ajoutée dans math3d.h (unit testé dans catch_mat.cpp) :

```
mat4 segment_transform(const vec3& world_start, const vec3& world_end)
{
    vec3 AB(world_end-world_start);
    float s = AB.norm(); // scale is just the length of input segment

    // If line is vertical, general transformation is singular (OH==0)
    // Rotation is around z-axis only and with angle pi/2
    if(fabs(AB.x())<0.0001f && fabs(AB.z())<0.0001f)
    {
        return mat4(0, -s, 0, world_start.x(),
                    s, 0,  0, world_start.y(),
                    0, 0,  s, world_start.z(),
                    0, 0,  0, 1);
    }
    // General transformation
    else
    {
        vec3 OH(AB.x(),0,AB.z()); // project AB on xz plane
        vec3 w(OH.normalized());  // unit vector along AB
        float k = AB.y()/s;       // = sin(theta) (theta angle around z-axis)
        float d = sqrt(1.0f-k*k); // = cos(theta)
        float l = w.x();          // = cos(phi) (phi angle around y-axis)
        float e = ((w.z()>=0.0f)?1.0f:-1.0f) * sqrt(1.0f-l*l); // = sin(phi)

        // Return pre-combined product of T*R*S matrices
        return mat4(s*l*d, -s*l*k, -s*e, world_start.x(),
                    s*k,   s*d,    0,    world_start.y(),
                    s*e*d, -s*e*k, s*l,  world_start.z(),
                    0,     0,      0,    1);
    }
}
```

Pour parvenir à ce résultat, j'ai décomposé la matrice affine recherchée en un produit d'une matrice de translation par une de rotation et par une d'échelle. En gros, on imagine qu'on essaye d'abord de rescale le segment unitaire pour lui donner la longueur du segment arbitraire, puis on le tourne selon z, puis selon y, puis on le translate. Les parties scaling et translation sont triviales. Il y a 2 remarques importantes concernant les rotations : * Dans l'évaluation de Ry, l'angle phi est connaissable au signe près, car il est obtenu par un arc-cosinus d'un produit scalaire de 2 vecteurs unitaires w et x. Considérons le produit vectoriel de w et x. Le produit scalaire du vecteur obtenu avec l'axe de rotation y donne le signe que l'on cherche (l'opération complète est donc un produit mixte). Quelques simplifications entrainent que ce signe est celui de w.z. * Plutôt que de calculer les angles via les fonctions inverse trigo et appliquer les fonctions directes sur ceux-ci, on peut se servir d'une relation bien utile pour simplifier les matrices : sin(acos(x)) = cos(asin(x)) = sqrt(1-x^2) L'expression de l'angle phi implique une fonction signe, et on se sert de la parité de sin et cos pour propager celui-ci (ou non), d'où les expressions de l et e. De fait, les angles

n'ont jamais besoin d'être calculés explicitement, on a directement les coefficients des matrices Ry et Rz (+rapide -d'erreur d'arrondi).

La transformation générale telle que décrite ici est singulière pour tous les segments verticaux. Donc je teste simplement si un vecteur est vectical à un epsilon près, et je retourne une transformation dégénérée plus simple si tel est le cas. J'ai pré-calculé à la main les produits matriciels et ma fonction retourne directement la combinaison totale. Je la pense assez optimisée et robuste.

Du coup, on peut faire :

```
void DebugRenderer::request_draw_segment(const math::vec3& world_start,
                                         const math::vec3& world_end,
                                         int ttl,
                                         const math::vec3& color)
{
    DebugDrawRequest request;
    request.type = DebugDrawRequest::SEGMENT;
    request.ttl = ttl;
    request.color = color;
    request.color[3] = 1.0f;
    request.model_matrix = math::segment_transform(world_start, world_end);
    draw_requests_.push_back(request);
}
```

Et dans la fonction render() on n'a plus qu'à parcourir la liste de draw requests, éliminer celles qui ont un TTL négatif, et afficher les autres (plutôt que de parcourir la liste une seconde fois avec std::remove_if après la boucle d'affichage) :

```
    auto it = draw_requests_.begin();
    while (it != draw_requests_.end())
    {
        // Remove dead requests
        bool alive = (--(*it).ttl >= 0);
        if (!alive)
        {
            draw_requests_.erase(it++);  // alternatively, it =
                draw_requests_.erase(it);
        }
        // Display the required primitive
        else
        {
            if((*it).type == DebugDrawRequest::SEGMENT)
            {
                // Get model matrix and compute products
                mat4 MVP(PV*(*it).model_matrix);

                line_shader_.send_uniform(H_("tr.m4_ModelViewProjection"), MVP);
                line_shader_.send_uniform(H_("v4_line_color"), vec4((*it).color));
                buffer_unit_.draw(SEG_NE, SEG_OFFSET);
            }
            ++it;
        }
    }
```

[21-12-18] Pas de repos pour les braves

Ray casting Screen -> NDC -> World

Le ray casting fonctionne (en tout cas la partie projection de rayon, donc l'essentiel). Le système *RayCaster* est un *Updatable* et un *Listener* configuré dans wcore.cpp pour écouter le canal "input.mouse.click" sur lequel émet *InputHandler* uniquement quand un bouton de la souris est cliqué. Sa fonction update() récupère les matrices de la caméra à chaque frame afin de calculer une matrice de déprojection qui est mise en cache pour toutes les requêtes de lancer de rayon. A chaque évenement souris, sa fonction cast_ray_from_screen() est appelée avec les coordonnées écran, calcule un rayon en coordonnées world et demande à la *RenderPipeline* de dessiner un segment grâce à la méthode mise au point hier.

Dans le détail, la déprojection se fait ainsi : * On prépare la matrice de déprojection en inversant la view-projection matrix à cette frame :

```
void RayCaster::update(const GameClock& clock)
{
    // Get camera view-projection matrix for this frame and invert it
    pCamera cam = SCENE.get_camera();
    const math::mat4& view = cam->get_view_matrix();
    const math::mat4& projection = cam->get_projection_matrix();
    eye_pos_world_ = cam->get_position();
    eye_pos_world_[3] = 1.0f;
    math::mat4 VP(projection*view);
    math::inverse(VP, unproj_);
}
```

- On commence avec des coordonnées écran $x_s, y_s \in [0, 1]$ avec $(0, 0)$ le coin inférieur gauche. On convertit en NDC :

```
void RayCaster::cast_ray_from_screen(const math::vec2& screen_coords)
{
math::vec4 coords(screen_coords);
coords *= 2.0f;
coords -= 1.0f;
coords[2] = 1.0f; // near
coords[3] = 1.0f;
```

  On veut se mettre en z=1 qui en NDC correspond au plan near (z=-1 correspond à far). La composante w est initialisée à 1.

- Ce point en coordonnées NDC est ensuite multiplié par la matrice de déprojection et subit un perspective divide :

```
math::vec4 wcoords(unproj_*coords);
wcoords /= wcoords.w();
```

- On calcule la direction du rayon par soustraction de la position de la caméra et normalisation :

```
math::vec4 direction(wcoords-eye_pos_world_);
direction.normalize();
```

- La donnée de wcoords, direction ainsi qu'une longueur totale forment un rayon, que je me contente d'afficher sous forme de segment pour l'instant : "'cpp pipeline_.debug_draw_segment(wcoords.xyz(), (wcoords+Camera::get_far()*direction).xyz(), 10*60, math::vec3(0,1,0));

"'

Ray/AABB intersection

J'ai implémenté l'algo décrit dans [1] pour effectuer des tests d'intersection entre un rayon et les AABBs des objets de la scène. J'ai supposé que ce serait plus simple pour un AABB qu'un OBB dans un premier temps (il semble que ce ne soit pas nécessairement le cas, voir [2]). Pour chaque dimension de l'espace, chaque paire opposée de plans d'un AABB forme une tranche (slab). Un rayon non parallèle au slab intersecte celui-ci en deux points (sur ses deux plans), l'un est proche, l'autre lointain. L'algo consiste à calculer les distances d'intersection pour chaque slab et à tenir à jour la valeur maximale des distances d'intersection proches (Tnear), ainsi que la valeur minimale des distances d'intersection lointaines (Tfar). Si Tfar C'est fait, grâce à la nouvelle fonction Scene::visit_model_first() qui visite uniquement le premier modèle à évaluer à true dans le prédicat du second argument. - La sélection gagnera en précision quand j'effectuerai des tests ray/OBB directement (si j'ai la garantie que ce n'est pas plus cher). -> C'est fait aussi, voir plus loin.

Sources :

```
[1] https://www.siggraph.org//education/materials/HyperGraph
/raytrace/rtinter3.htm
[2] http://www.opengl-tutorial.org/miscellaneous/clicking-on-objects/
picking-with-custom-ray-obb-function/
```

[23-12-18]

Ray/OBB intersection

En réfléchissant juste 2 secondes, j'ai compris que le code d'intersection ray/AABB pouvait être réutilisé. Pour réaliser le test d'intersection ray/OBB, il suffit de passer le rayon dans l'espace modèle et de lancer un test ray/AABB. Une mise à l'échelle des résultats de collision est nécessaire en cas de hit. Toute la viande de ray_collides_AABB() est remballée sous le nom ray_collides_extent() qui prend un extent en argument (extent_t = std::array). ray_collides_AABB() et ray_collides_OBB() utilisent cette fonction en sous-main.

```
bool ray_collides_OBB(const Ray& ray, std::shared_ptr<Model> pmdl,
    RayCollisionData& data)
{
    // Transform ray to model space
    bool ret = ray_collides_extent(ray.to_model_space(pmdl->get_model_matrix()),
        pmdl->get_mesh().get_dimensions(), data);
    // Rescale hit data
    if(ret)
    {
        float scale = pmdl->get_transformation().get_scale();
        data.near *= scale;
        data.far *= scale;
    }
    return ret;
}
```

La fonction Ray::to_model_space() prend une matrice modèle en argument, calcule son inverse (math::inverse_affine() optimale pour de telles matrices) et applique la matrice obtenue sur l'origine et l'extrémité du rayon, avant de recalculer la direction normalisée. Un nouveau rayon est généré, cette fois dans l'espace modèle, où les axes sont alignés avec l'OBB…

La sélection lance maintenant des tests ray/OBB et stop au premier hit, alors que la scène est parcourue front to back, l'objet sélectionné est donc le plus proche possible de la cam. Tout semble bien fonctionner. - La sélection utilise un membre weak_ptr de la scène. Elle est naturellement invalidée quand l'objet est détruit (comme après un déchargement de chunk). - On pourrait aussi imaginer retourner toute la liste des objets qui intersectent le rayon,

sélectionner le premier, et laisser l'utilisateur avancer ou reculer la sélection, afin d'améliorer l'opération sur un gros cluster de modèles.

GLEW rant

https://www.reddit.com/r/opengl/comments/3m28x1/glew_vs_glad/

- It doesn't generate or handle functions properly. One example is VAO functions. The functions are core in GL3+, and also available if the GL_ARB_vertex_array_object extension is available. However because of the crappy way GLEW's headers have been generated, by default it will only load the function pointers if GL_ARB_vertex_array_object is available – even if a GL3 context is used! This completely breaks everything on platforms like OS X where it exposes GL3 and the VAO functions, but doesn't expose the GL_ARB_vertex_array_object extension. (There are many similar examples.)
- It doesn't load extensions properly in GL3+. It tries to use a removed function (glGetString(GL_EXTENSIONS), rather than glGetStringi(GL_EXTENSIONS, index)), which means no function pointers for extensions will be loaded at all unless glewExperimental is used.
- glewExperimental is a totally broken idea. It just tries to load the function pointer of all function names from all extensions, and bases its "reported extension support" on whether the functions exist on the system. Not only is this very slow, it also completely breaks when the system has more functions in its GL library than the actual context supports. For example in OS X if you use a legacy GL2 context, it won't support GL3 nor several modern extensions. However GLEW thinks they're supported because OpenGL.framework has those function pointers. So it will report GL_ARB_vertex_array_object (and others) as being supported when they're not and will cause runtime errors if you have code that queries GLEW for whether they're supported.

[28-12-18] RK integrator

Je suis en train de prototyper un intégrateur Runge-Kutta-Nyström d'ordre 4 (RKN4) pour la simulation physique. Pour l'instant j'intègre uniquement la position et la vitesse du centre de gravité d'un solide, la partie orientation est autrement plus poilue et verra le jour un peu plus tard.

Je me base essentiellement sur [1] p.5 (il y a une erreur pour le calcul de k3, la dépendance est en k2 et non en k1).

Définissons l'état du solide via la structure suivante :

```
struct RigidBodyState
{
    math::vec3 position; // Center of gravity position
    math::vec3 velocity; // C.o.G instant velocity
    float mass = 1.f;    // Total object mass
};
```

Pour l'instant, considérons des forces qui s'appliquent sur le solide à chaque instant (non-impulse). On fait l'hypothèse qu'on peut les écrire en fonction de la position, de la vitesse, de la masse et du temps :

```
typedef std::function<math::vec3 (const math::vec3&, const math::vec3&, float,
    float)> force_t;

static force_t gravity = [](const math::vec3& pos, const math::vec3& vel, float m,
    float t)
{
    return m*math::vec3(0.f, -9.81f, 0.f);
};
static force_t drag = [](const math::vec3& pos, const math::vec3& vel, float m,
    float t)
{
    return -5.0f*vel.norm2()*vel.normalized();
```

```
};
static force_t total_force = [](const math::vec3& pos, const math::vec3& vel,
    float m, float t)
{
    return gravity(pos,vel,m,t) + drag(pos,vel,m,t);
};
```

Alors un pas d'intégration suit l'algorithme :

```
RigidBodyState integrate_RKN4(const RigidBodyState& last, force_t force, float t,
    float dt)
{
    float mass_inv = 1.0f/last.mass;
    math::vec3 k1 = mass_inv * force(last.position, last.velocity, last.mass, t);
    math::vec3 k2 = mass_inv * force(last.position + 0.5f*dt * last.velocity +
        0.125f*dt*dt * k1,
                                     last.velocity + 0.5f*dt * k1, last.mass, t +
                                        0.5f*dt);
    math::vec3 k3 = mass_inv * force(last.position + 0.5f*dt * last.velocity +
        0.125f*dt*dt * k2,
                                     last.velocity + 0.5f*dt * k2, last.mass, t +
                                        0.5f*dt);
    math::vec3 k4 = mass_inv * force(last.position +      dt * last.velocity +
        0.5f*dt*dt * k3,
                                     last.velocity +      dt * k3, last.mass, t +
                                            dt);
    RigidBodyState next;
    next.position = last.position + dt*last.velocity + dt*dt/6.0f * (k1+k2+k3);
    next.velocity = last.velocity + dt/6.0f * (k1 + 2.0f*k2 + 2.0f*k3 + k4);
    next.mass     = last.mass;
    return next;
}
```

On peut alors définir un état initial et intégrer celui-ci sous l'effet des forces de gravité et de trainée cumulées :

```
    RigidBodyState body1;
    body1.position = vec3(0,10,0);
    body1.velocity = vec3(1,0,0);
    body1.mass = 10.f;

    float t  = 0.f;
    float dt = 1.0f/60.0f;
    for(int ii=0; ii<60*5; ++ii)
    {
        body1 = integrate_RKN4(body1, total_force, t, dt);
        t += dt;
    }
```

Afin de plotter le résultat, j'utilise un pipe vers GnuPlot (voir [3]). Son utilisation nécessite de linker l'exécutable avec -lboost_iostreams, -lboost_system et -lboost_filesystem. On doit include gnuplot-iostream.h qui se trouve dans vendor/gnuplot. Gnuplot nécessite 4 colonnes pour pouvoir plotter une courbe comme une suite de vecteurs (x,y,dx,dy). Il faut donc former ces vecteurs lors de l'itération :

```
    std::vector<std::tuple<float, float, float, float>> plot_points;

    float t  = 0.f;
    float dt = 1.0f/60.0f;
    for(int ii=0; ii<60*5; ++ii)
```

```
    {
        float x = body1.position.x();
        float y = body1.position.y();

        body1 = integrate_RKN4(body1, total_force, t, dt);
        //std::cout << body1 << std::endl;
        t += dt;

        float dx = body1.position.x()-x;
        float dy = body1.position.y()-y;
        plot_points.push_back(std::make_tuple(x,y,dx,dy));
    }
```

Puis on utilise le pipe pour communiquer les données à GnuPlot :

```
    Gnuplot gp;

    // Don't forget to put "\n" at the end of each line!
    gp << "set xrange [0:1]\nset yrange [0:10]\n";
    // '-' means read from stdin.  The send1d() function sends data to gnuplot's
       stdin.
    gp << "plot '-' with vectors title 'traj'\n";
    gp.send1d(plot_points);
```

L'état initial fait partir l'objet de l'altitude y=10 avec la vitesse horizontale vx=1. On obtient une belle courbe caractéristique du mouvement freiné d'un objet dans un fluide visqueux. Mon premier test plus simple consistait à lacher l'objet verticalement initialement au repos, et a bien vérifier que j'obtenais une vitesse verticale vy=-9.81 au bout d'une seconde (60 pas d'intégration à dt=1/60), ce qui était le cas.

Sources :


```
[1] Rigid body dynamics using Euler's equations, Runge-Kutta and
quaternions. Indrek Mandre feb. 26, 2008
http://www.mare.ee/indrek/varphi/vardyn.pdf
[2] Geometric Integration of Quaternions. Michael S. Andrle,
John L. Crassidis. University at Buffalo, State University of New York,
Amherst, NY, 14260-4400
http://ancs.eng.buffalo.edu/pdf/ancs_papers/2013/geom_int.pdf
[3] http://stahlke.org/dan/gnuplot-iostream/
```


[29-12-18]

- Je viens de corriger une erreur merdique dans mon code de quaternions, d'invalider un unit test et d'optimiser le calcul de rotations de vecteurs par quaternion.

Avant :

```
vec3 Quaternion::rotate(const vec3& vector) const
{
    return get_conjugate().get_rotation_matrix()*vector;
}
```

Après :

```
vec3 Quaternion::rotate(const vec3& vector) const
{
    //return get_rotation_matrix()*vector;
```

```
    return ((*this)*Quaternion(vector)*get_conjugate()).get_as_vec().xyz();
}
```

La seule raison pour laquelle j'utilisais get_conjugate() au lieu du quat lui-même était pour coller à ce que me sortait Matlab lors du unit testing. Mais Matlab semble effectuer des rotations CW au lieu de CCW. J'utilise maintenant la vraie formule, qui consiste à augmenter le vecteur pour en faire un quaternion avec une composante scalaire nulle, et à le prendre en sandwich entre le quat d'orientation et son conjugué (voir [1]) :

```
r = (r_x, r_y, r_z) -> R = (r_x, r_y, r_z, 0)
calculer q*r*q_  avec q_ = (-q_x, -q_y, -q_z, q_w)
```

Le vecteur retourné est simplement la partie vectorielle du quaternion obtenu (on peut montrer que la composante scalaire est toujours nulle).

Pour la rotation inverse on fait :

```
vec3 Quaternion::rotate_inverse(const vec3& vector) const
{
    // return get_conjugate().get_rotation_matrix()*vector;
    return (get_conjugate()*Quaternion(vector)*(*this)).get_as_vec().xyz();
}
```

Voici du code qui en l'état me confirme que je fais les choses de manière cohérente :

```
    vec3 v1(1,0,0);
    vec3 v2(0,0,1);
    vec3 v3(0,0,-1);
    vec3 v4(0,1,0);
    vec3 v5(0,-1,0);
    quat q1(0.0, 0.7071, 0.0, 0.7071); // == quat q1(0, 90, 0);
    quat q2(0.0, 0.0, 0.7071, 0.7071); // == quat q2(90, 0, 0);

    q1.normalize();
    q2.normalize();

    mat4 m1,m2;
    math::init_rotation_euler(m1, 0, TORADIANS(90), 0);
    math::init_rotation_euler(m2, TORADIANS(90), 0, 0);

    std::cout << "q1: rot +90 around y axis" << std::endl;
    std::cout << v1 << " -> " << q1.rotate(v1) << std::endl;
    std::cout << v2 << " -> " << q1.rotate(v2) << std::endl;
    std::cout << v3 << " -> " << q1.rotate(v3) << std::endl;
    std::cout << "q2: rot +90 around z axis" << std::endl;
    std::cout << v1 << " -> " << q2.rotate(v1) << std::endl;
    std::cout << v4 << " -> " << q2.rotate(v4) << std::endl;
    std::cout << v5 << " -> " << q2.rotate(v5) << std::endl;

    std::cout << "q1.rot_matrix: rot +90 around y axis" << std::endl;
    std::cout << v1 << " -> " << q1.get_rotation_matrix()*v1 << std::endl;
    std::cout << v2 << " -> " << q1.get_rotation_matrix()*v2 << std::endl;
    std::cout << v3 << " -> " << q1.get_rotation_matrix()*v3 << std::endl;
    std::cout << "q2.rot_matrix: rot +90 around z axis" << std::endl;
    std::cout << v1 << " -> " << q2.get_rotation_matrix()*v1 << std::endl;
    std::cout << v4 << " -> " << q2.get_rotation_matrix()*v4 << std::endl;
    std::cout << v5 << " -> " << q2.get_rotation_matrix()*v5 << std::endl;

    std::cout << "m1: rot +90 around y axis" << std::endl;
    std::cout << v1 << " -> " << m1*v1 << std::endl;
```

```
    std::cout << v2 << " -> " << m1*v2 << std::endl;
    std::cout << v3 << " -> " << m1*v3 << std::endl;
    std::cout << "m2: rot +90 around z axis" << std::endl;
    std::cout << v1 << " -> " << m2*v1 << std::endl;
    std::cout << v4 << " -> " << m2*v4 << std::endl;
    std::cout << v5 << " -> " << m2*v5 << std::endl;
```

```
  q1: rot +90 around y axis
  (1, 0, 0) -> (0, 0, -1)
  (0, 0, 1) -> (1, 0, 0)
  (0, 0, -1) -> (-1, 0, 0)
  q2: rot +90 around z axis
  (1, 0, 0) -> (0, 1, 0)
  (0, 1, 0) -> (-1, 0, 0)
  (0, -1, 0) -> (1, 0, 0)
  q1.rot_matrix: rot +90 around y axis
  (1, 0, 0) -> (-1.19209e-07, 0, -1)
  (0, 0, 1) -> (1, 0, -1.19209e-07)
  (0, 0, -1) -> (-1, 0, 1.19209e-07)
  q2.rot_matrix: rot +90 around z axis
  (1, 0, 0) -> (-1.19209e-07, 1, 0)
  (0, 1, 0) -> (-1, -1.19209e-07, 0)
  (0, -1, 0) -> (1, 1.19209e-07, 0)
  m1: rot +90 around y axis
  (1, 0, 0) -> (-4.37114e-08, 0, -1)
  (0, 0, 1) -> (1, 0, -4.37114e-08)
  (0, 0, -1) -> (-1, 0, 4.37114e-08)
  m2: rot +90 around z axis
  (1, 0, 0) -> (-4.37114e-08, 1, 0)
  (0, 1, 0) -> (-1, -4.37114e-08, 0)
  (0, -1, 0) -> (1, 4.37114e-08, 0)
```

- TODO: [ ] Noter que l'initialisation des quats par angles de Tait-Bryan suppose des arguments en degrés, tandis que la même initialisation pour les matrices les suppose en radians. Il faudra corriger cette inconsistence.
- J'essaye de prototyper l'intégration de l'orientation d'un solide. J'utilise une méthode d'Euler semi-implicite taillée pour prendre en compte le couple gyroscopique (effet de précession, bien souvent omis par les gamedevs pour des raisons de stabilité numérique) (voir [2]). Le calcul de la vitesse angulaire se fait en coordonnées locales (comme ça le tenseur d'inertie est constant), un "vecteur résiduel" et un Jacobien sont calculés et utilisés dans un pas de Newton-Raphson pour mettre à jour la vitesse angulaire. La nouvelle orientation (quaternion) est calculée depuis la vitesse angulaire (voir [3]). Pour l'instant, je calcule le moment cinétique (L = I * omega) à chaque pas d'intégration, et je constate qu'il n'est pas constant alors que je n'applique aucun couple sur le système, donc j'ai nécessairement un souci qq part.

Sources :

```
[1] http://www.cs.cmu.edu/afs/cs.cmu.edu/user/spiff/www/moedit99/expmap.pdf
[2] https://www.gdcvault.com/play/1022196/Physics-for-Game-Programmers
-Numerical
[3] https://fgiesen.wordpress.com/2012/08/24/quaternion-differentiation/
```

[30-12-18] Intern strings & H_ macro

Plutôt que de persister à utiliser le define **PRESERVE_STRS** qui est complètement pété, j'ai choisi une nouvelle approche. J'ai écrit un petit utilitaire nommé "internstr" (host app), qui parse les sources (.h et .cpp) à la recherche de macros H_("…") (j'ai viré HS_ et hstr_t). Les string et les hash qui leur correspondent sont ensuite

stockés dans un fichier XML (config/dbg_intern_strings.xml). Le nouveau foncteur singleton *InternStringLocator* de intern_string.h parse le fichier XML en runtime et récupère cette table. Un alias de *InternStringLocator* du nom de HRESOLVE peut être appelé (partout où wtypes.h est inclu) afin de résoudre les hash en runtime ! Si un hash ne peut être résolu, la chaîne "???" est renvoyée à la place. Donc il suffit de lancer l'utilitaire quand on crée une nouvelle string interne via H_ et le hash sera automatiquement solvable en runtime.

Certains systèmes (dont *MaterialFactory*) génèrent des intern strings en runtime et doivent donc soumettre celles-ci à HRESOLVE via InternStringLocator::add_intern_string().

Le seul truc qui m'a cassé les burnes est l'utilisation d'un regex pour matcher toutes les occurrences dans un fichier.

L'application se localise et remonte au dossier root, puis cherche les chemins d'accès vers les sources (inc_path_ et src_path_). Chaque fichier est alors parsé :

```
    for(const auto& entry: fs::directory_iterator(inc_path_))
        parse_entry(entry);
    for(const auto& entry: fs::directory_iterator(src_path_))
        parse_entry(entry);
```

La fonction parse_entry() est la suivante :

```
static std::regex hash_str_tag("H_\\(\"(.+?)\"\\)");
static std::map<hash_t, std::string> intern_strings_;
static void parse_entry(const fs::directory_entry& entry)
{
    // * Copy file to string

    // ...

    // * Match string hash macros and update table
    std::regex_iterator<std::string::iterator> it(source_str.begin(),
       source_str.end(), hash_str_tag);
    std::regex_iterator<std::string::iterator> end;

    while(it != end)
    {
        std::string intern((*it)[1]); // The intern string
        unsigned long long hash_intern = H_(intern.c_str()); // Hashed string

        if(intern_strings_.find(hash_intern) == intern_strings_.end())
            intern_strings_.insert(std::make_pair(hash_intern, intern));

        ++it;
    }
}
```

Le regex_iterator peut être déréférencé pour retourner un std::smatch. L'élément 1 du smatch est la première capture : l'argument de la macro.

Noter que le regex ignore les cas d'utilisation runtime de la macro H_, car des guillemets doivent être présents pour qu'il y ait match.

J'ai ensuite modifié parse_entry() pour détecter les collisions de hash. Si un hash est déjà présent dans le tableau, alors la string correspondante est comparée à la nouvelle string qui a produit le même hash. Si les deux strings sont différentes, c'est qu'il y a une collision, le programme génère un warning et demande l'appui sur la touche ENTER avant de poursuivre.

[01-01-19]

J'ai réglé TOUS les problèmes que j'avais avec les AABB/OBB, qui sont maintenant parfaitement calés et optimaux. Il y a juste une distinction à faire entre les modèles dont le mesh est centré sur l'origine en model space et ceux qui ont un ymin=0. Si le modèle n'est pas centré, il faut translater son OBB verticalement, puisque j'utilise les vertices d'un cube centré dans la fonction OBB::update(). La fonction AABB::update() utilise maintenant l'OBB du modèle parent pour éliminer les calculs inutiles. Tout est beaucoup plus clair et hack-free.

[02-01-19] Du gros Octree qui tache

Je suis en train de prévoir la broad phase du moteur physique. J'ai dev un début de classe *Octree* dans octree.hpp. Pour l'instant je ne peux classer que des points, la classe devra évoluer pour classer des objets avec des AABBs. La classe est templatée par une structure contenant une ou plusieurs listes d'objets. Cette structure doit définir différentes fonctions pour être utilisable dans *Octree*. Je ne suis pas totalement satisfait de cette approche, donc je tairai le détail pour le moment. L'octree possède des membres parent_ et children_ de type Octree* . L'octree root (qui englobera le niveau entier) possède un parent_ nullptr, et les leaves ont leur children_ nullptr. Un autre membre important est bounding_region_ (qui contient essentiellement un math::extent_t et un membre mid_point initialisé RAII) qui fixe les limites spatiales d'un noeud donné de l'arbre. La partie "compliquée" consiste à subdiviser récursivement l'espace en 8 octants à chaque fois que c'est nécessaire, et à répartir le contenu d'une cellule à subdiviser dans les 8 octants enfants. La viande se trouve donc dans Octree::subdivide() qui prend un prédicat et un Octree* en arguments. Le prédicat sert à évaluer depuis l'extérieur de la classe si la cellule spatiale courante doit encore être subdivisée :

```
1    typedef std::function<bool(const content_t&, const BoundingRegion&)>
        subdivision_predicate_t;
```

Le deuxième argument permet la chaîne de récursivité.

A chaque niveau de récursion, on teste si on doit encore subdiviser en se servant du prédicat. On peut vouloir contraindre la taille minimale des cellules ou le nombre d'objets max qu'elles peuvent contenir par exemple (je fais les deux). Si on ne peut pas subdiviser, alors on a notre condition d'arrêt. Si on peut subdiviser, alors on commence par allouer et initialiser les 8 octants enfants. Puis on calcule les limites spatiales des octants, à partir du centre de la cellule courante et de ses propres limites. Ensuite le contenu de la cellule courante est dispatché dans les octants, puis les enfants sont subdivisés à leur tour…

Pour l'instant, comme je ne classe que des points pour ce prototype d'octree, tous mes objets finissent nécessairement dans les feuilles de l'octree. Donc à chaque niveau de récursion, je vide complètement la liste courante dans les octants. Plus tard, quand je devrai classer des bounding boxes, il faudra déterminer si les octants sont suffisamment larges pour accueillir celles-ci, le cas échéant l'objet restera dans le noeud parent (ne sera pas retiré de sa liste). La structure actuelle s'appelle un Point Octree, et reste intéressante à conserver (utile pour les algos de particules type flocking…). Il faudrait que je puisse templater l'octree avec la classe d'objets à partitionner, ce serait plus intelligent que de tout foutre en l'air pour remplacer par des AABBs.

- NOTE: Mes *AABB* ont pour le moment un membre Model& (qui leur sert à récupérer l'*OBB* du parent pour l'update), ça va être gênant pour en faire une utilisation avec les *Entity* plus tard. -> C'est réglé, gros refactor des bounding boxes.

-> Quand j'utiliserai l'octree dans mon moteur, il faudra veiller à ce que les chunks puissent être contenus dans des cellules à leur taille. D'une certaine manière, les chunks deviendront une unité d'insertion pour l'octree. -> J'ai l'intention d'utiliser un octree pour la géométrie statique, et un autre pour la géométrie dynamique. Seul le second aura besoin d'être reconstruit en temps réel, et comme a priori il y aura moins d'objets dynamiques que statiques, je pense faire une économie CPU. -> Je pense aussi me servir de l'octree pour accélérer (O(n)->O(log(n))) le frustum culling (genre, on peut skip toute une branche dont la racine n'est pas visible). -> Plus tard on pourra tirer partie de cette structure pour du LoD dynamique sur le terrain, ça serait super sexy.

- TODO: [x] Gérer les bounding boxes -> Move object to child list iif child bounds are large enough -> Only delete moved objects from the list (use delete list) [x] Ecrire une fonction d'insertion en lazy init à la [1] ->

On feed une liste d'objets à rajouter dans l'octree -> On insère la liste en une seule fois, plutôt que d'avoir à effectuer une reconstruction partielle/totale de l'octree à chaque insertion.

Sources :

```
1 [1] https://www.gamedev.net/articles/programming/general-and-gameplay
-programming/introduction-to-octrees-r3529/
```

[05-01-19]

REFACTOR Bounding boxes

- Une nouvelle structure *BoundingRegion* cumule les 2 représentations possibles d'un volume spatial cubique : un math::extent ET un couple de vecteurs pour le centre (mid_point) et les demi-dimensions (half). Comme mes algos utilisent les 2 représentations, j'ai pensé que l'overhead valait le coup.
- La classe *OBB* qui possède un *BoundingRegion* en model space, est construite depuis un math::extent d'un modèle parent et un booléen qui traduit si le mesh est centré ou non sur l'origine en model space. La fonction update() prend une matrice modèle en argument (celle du parent).
- La classe *AABB* possède un *BoundingRegion* en world space, et est updatée via un objet *OBB* (celui du parent).
- J'ai clarifié et uniformisé la sémantique pour les tests de collisions entre mes primitives volumétriques :
  - intersects() permet de savoir si 2 primitives sont en collision.
  - contains() permet de savoir si une primitive en contient une autre.
    - En particulier, contains() et intersects() sont équivalentes pour un point uniquement.

Octree

J'ai méga avancé dans le développement de l'octree.

Je différencie maintenant les noeuds de type *OctreeNode<>* de l'octree lui-même *Octree<>*. Ca me permettra de manipuler la racine de l'arbre plus facilement, ce qui est essentiel pour pouvoir étendre et contracter l'octree quand un nouvel objet est inséré hors des limites spatiales de la racine ou respectivement qu'un objet est détruit et que 7 des 8 octants de la racine sont vides (à venir). La classe *Octree<>* possède pour l'instant un unique membre root_ et un ensemble de méthodes dont la plupart se contentent d'appeler les méthodes d'*OctreeNode<>*. La structure supporte l'insertion de données et leur suppression, ainsi qu'une méthode de query qui permet de visiter les noeuds compris dans un volume spatial spécifié en argument.

Définitions : * *Primitive* : Le type d'objet utilisé dans l'arbre pour localiser les données. Il peut s'agir d'un point (math::vec3), d'un volume cubique (*BoundingRegion* ou *AABB*), d'une sphère… * *User data* : Le type de données associées aux primitives dans l'arbre. Il peut s'agir d'un entity ID, d'un pointeur vers un objet du jeu… C'est nécessairement un type comparable. * *Data* : Cette structure encapsule une primitive et un objet User data (plus d'autres flags éventuels pour les besoins de l'octree).

Insertion et propagation

Pour insérer des données, on les envoie d'abord dans la liste de données de la racine. Puis une méthode propagate() est appelée, qui fait descendre les données dans l'arbre récursivement. Pour chaque niveau de récursion : - On vérifie si la cellule courante doit être subdivisée (s'il y a trop de données au niveau courant et qu'on n'a pas atteint la profondeur maximale). Si tel est le cas, alors on subdivise et on poursuit l'algo. - Si on n'est pas dans une feuille { - Pour chaque objet à ce niveau : - On calcule l'indice de l'octant le plus à même d'accueillir l'objet - On vérifie que l'objet peut rentrer dans l'octant. Si c'est le cas on l'y envoie, sinon il reste sur place. - On propage les données des enfants } - Sinon { - On stop la récursion en retournant. } Noter que les données sont accompagnées d'un flag

qui permet de savoir si elles ont atteint le niveau idoine dans l'arbre. Une donnée placée ne sera plus testée. Si la cellule doit être subdivisée, alors ce flag est invalidé pour toutes les données courantes. Si on est dans une feuille alors les données sont nécessairement à destination.

Pour calculer l'indice du meilleur octant pour envoyer les données, j'ai fait un truc malin inspiré de [1]. L'indice de mes octants correspond aux choix cumulés d'un arbre de décision binaire :

```
X: x>x_c ? 1 : 0
Z: z>z_c ? 2 : 0
Y: y>y_c ? 4 : 0

                  X?
                 n/\y
                 /  \
                0    1
               Z?    Z?
               /\    /\
              /  \  /  \
             0   2 0   2
            Y?  Y? Y?  Y?
            /\  /\ /\  /\
           /  \/  \/  \/  \
           0  40  40  40  4
```

Par exemple, pour l'octant droit (x < xc) vers l'avant (z > zc) supérieur (y > yc), l'indice vaut 0 + 2 + 4 = 6. Donc pour calculer l'indice du meilleur octant je fais :

```
template <OCTREE_NODE_ARGLIST>
inline uint8_t OCTREE_NODE::best_fit_octant(const PrimitiveT& primitive)
{
    // Octants are arranged in a binary decision tree fashion
    // We can use this to our advantage
    math::vec3 diff(center(primitive) - bounding_region_.mid_point);
    return (diff.x()<0 ? 0 : 1) + (diff.z()<0 ? 0 : 2) + (diff.y()<0 ? 0 : 4);
}
```

Noter qu'il doit exister une fonction center(const PrimitiveT&) qui renvoie le centre de la primitive. Pour un point, c'est le point lui-même, pour un AABB, c'est son… centre, pour une sphère… son centre également etc. Un gros avantage est que je vais pouvoir utiliser cette fonction pour l'extension automatique de l'octree (et surement à d'autres endroits).

Suppression et merge

Pour faire référence à un objet à supprimer dans l'arbre, on utilise les user data communiquées avec la primitive lors de l'insertion. Le type UserDataT doit être comparable (définir bool operator==()). En pratique on peut former une structure du genre :

```
struct UData
{
    float value;
    int key;

    bool operator==(const UData& other)
    {
        return key == other.key;
    }
};
```

Avec key un hash des données membres ou bien un unique id fourni par le moteur. Il est capital de s'assurer que cette clé est bien unique dans l'arbre. C'est un des points faibles de mon implémentation, mais je préfère ne pas avoir à calculer des hash dans l'octree pour accélérer les calculs.

L'algo de suppression est un depth first traversal qui va simplement comparer les données stockées à celle en argument et supprimer l'objet quand il est trouvé. Cependant, il peut arriver qu'une suppression libère suffisamment de place dans les octants d'un niveau donné pour que ces octants puissent être refondus au niveau courant (merge).

Pour chaque niveau de récursion on fait : - Pour chaque objet courant - Si l'objet compare on le vire et on retourne true - Si on n'est pas dans une feuille - Pour chaque octant enfant - Si on parvient à supprimer l'objet dans un des octants - Si on peut merge on merge - On retourne true - On retourne false

Pour savoir si on peut merge : - Si on est une feuille on ne peut pas (return false) - On initialise un compte total avec le nombre d'objets courants - Pour chaque octant - Si l'octant n'est pas une feuille alors il y a nécessairement trop d'objets sous le niveau courant donc on retourne false - On incrémente le compte total du nombre d'objets de l'octant - Si le compte total dépasse la capacité maximale on retourne false - On retourne true

Le merge est un simple splice() des listes d'objets des octants dans celle du niveau courant, et une suppression des enfants.

Query, traversal, visit

Je suis assez fier de ma fonction de visite :

```
template <OCTREE_NODE_ARGLIST>
template <typename RangeT>
void OCTREE_NODE::traverse_range(const RangeT& query_range,
                                 DataVisitorT visit,
                                 OctreeNode* current)
{
    // * Initial condition
    if(current == nullptr)
        current = this;

    // * Stop condition
    // Check bounding region intersection, return if out of range
    if(!query_range.intersects(current->bounding_region_))
        return;

    // * Visit objects within range at this level
    for(auto&& data: current->content_)
        if(query_range.intersects(data.primitive))
            visit(data);

    // * Walk down the octree recursively
    if(!current->is_leaf_node())
        for(int ii=0; ii<8; ++ii)
            traverse_range(query_range, visit, current->children_[ii]);
}
```

RangeT peut être n'importe quel objet volumétrique tant qu'il définit une fonction intersects() avec un objet *BoundingRegion* ET avec la primitive. Cet objet peut être un bête *BoundingRegion* ce qui permet de visiter uniquement les objets contenus dans cette région cubique, mais ce qui est super sexy c'est que RangeT peut être un objet *FrustumBox*, ce qui permet de faire un visibility traversal grâce au frustum de la caméra !

Sources :

```
[1] https://github.com/Nition/UnityOctree/blob/master/Scripts
/PointOctreeNode.cs
```

[06-01-19] Binary decision trees FUCK YEAH

J'ai posé toutes les maths pour l'extension de l'octree. Mon approche a été de raisonner systématiquement dans le cas 2D avec un quadtree, d'intuiter des formules reliant ce que je dois calculer à des opérations binaires sur les indices des octants (en tirant partie du fait qu'ils sont construits via un arbre de décision binaire), puis d'étendre ces formules au cas 3D.

Iterative subdivision

En particulier, j'ai pu réécrire très proprement ma fonction de subdivision comme suit :

```
template <OCTREE_NODE_ARGLIST>
void OCTREE_NODE::subdivide()
{
    // * Allocate children nodes
    children_ = new OCTREE_NODE*[8];

    // * Set children properties
    for(int ii=0; ii<8; ++ii)
    {
        children_[ii] = new OCTREE_NODE;
        children_[ii]->parent_ = this;
        children_[ii]->depth_  = depth_ + 1;
    }

    // * Subdivide bounding region into 8 octants
    const math::vec3& center = bounding_region_.mid_point;

    // Arrange octants according to a binary decision tree :
    for(uint8_t ii=0; ii<8; ++ii)
    {
        math::vec3 new_half   = 0.5f*bounding_region_.half;
        math::vec3 new_center = center
                              + math::vec3(((ii&1)?1.f:-1.f)*new_half.x(),
                                           ((ii&4)?1.f:-1.f)*new_half.y(),
                                           ((ii&2)?1.f:-1.f)*new_half.z());
        children_[ii]->bounding_region_ = BoundingRegion(new_center, new_half);
    }
}
```

Simplement, j'ai remarqué que dans le cas 2D, lors d'une subdivision, on peut toujours calculer la position des centres des quadrants enfants comme la somme vectorielle du centre du quadrant parent avec un offset dépendant de l'indice. Cet offset est toujours la demi-dimension des quadrants enfants, mais chaque composante possède un signe qui dépend d'un masquage binaire sur l'indice. Voir le cahier pour les maths, c'est over-chiant à écrire en céfran. Mais intuitivement, si on peut additionner les évaluations des décisions binaires pour obtenir un indice, alors il existe une opération inverse qui depuis un indice permet de remonter aux évaluations des décisions binaires par masquage. Dans le cas 3D avec l'octree, il y a simplement une décision binaire supplémentaire, les maths restent les mêmes.

[12-01-19]

REFACTOR Game systems

J'ai refactor une bonne partie du moteur afin d'homogénéiser l'utilisation et l'accès aux différents systèmes.

La nouvelle class *GameSystem* de game_system.h subsume *Updatable* et forme l'interface des systèmes de jeu. C'est un *Listener* qui peut donc recevoir des évenements, cette classe peut aussi être updatée et générer un widget pour l'éditeur. Les *GameSystem* sont enregistrés dans un conteneur *GameSystemContainer* dont un exemplaire est possédé par *GameLoop*. *GameSystemContainer* enregistre les systèmes avec un nom (hash_t), et peut être itéré (via begin() et end(), ce qui le rend utilisable dans un range for). En interne, *GameSystemContainer* enregistre les systèmes dans deux conteneurs : une liste ordonnée (c'est elle qui est itérable) et une map pour l'association aux noms de systèmes.

Chaque *GameSystem* possède un pointeur vers son *GameSystemContainer* parent. De fait il fut aisé d'implémenter une méthode GameSystem::locate(hash_t) qui permet à n'importe quel *GameSystem* d'obtenir un pointeur vers un autre *GameSystem* du même conteneur :

```
    template <typename T>
    inline T* locate(hash_t system_name)
    {
        return static_cast<T*>(parent_container_->get_system_by_name(system_name));
    }
```

La résolution des noms équivaut à un accès map O(log(N)) et il convient donc de résoudre le nom en pointeur une seule fois avant de s'en servir pour éviter l'overhead.

Grâce à cette méthode, la *Scene* qui est devenu un *GameSystem* n'avait plus de raison d'être un singleton, je lui ai donc retiré cet héritage.

Voici comment on peut obtenir un pointeur vers la scène depuis un autre *GameSystem* :

```
    Scene* pscene = locate<Scene>(H_("Scene"));
```

La classe *Engine* n'a plus qu'à enregistrer les systèmes via eimpl_->game_loop->register_game_system(), la classe *GameSystemContainer* gère le reste. Il sera donc aisé de prévoir des systèmes utilisateur pouvant être enregistrés au même niveau que les autres.

Consume events

La classe *Informer* a été modifiée afin de permettre aux *Listener* de consommer les évènements. Un évènement consommé par un délégué ne sera pas propagé aux autres délégués associés au même channel. Pour celà, il faut tenir compte de 2 choses : - Les foncteurs WpFunc qui peuvent être passés en argument à subscribe() sont maintenant booléens, ils retournent true quand l'event doit être propagé, et false quand l'event doit être consommé. - L'ordre de souscription détermine entièrement la priorité d'un *Listener*. Si A souscrit avant B au channel "input.keyboard" et que A consomme l'event dans sa fonction delegate, alors B ne recevra jamais l'event.

En l'état, les *GameSystem* souscrivent aux events via leur fonction GameSystem::init_events() qui est appelée dans GameSystemContainer::register_game_system(). **Donc l'ordre d'enregistrement des systèmes par Engine::Init() détermine à la fois l'ordre des updates ET la préséance de ces systèmes vis-à-vis de la consommation des évènements**. Ce comportement pourra être considéré comme non souhaitable à l'avenir et être amené à changer.

Exemples illustrant l'intérêt de cette fonctionnalité : * Le (futur) GUI (non debug) du jeu doit pouvoir consommer les events, de sorte qu'un clic sur un bouton par exemple, ne soit pas propagé au reste des systèmes.

[13-01-19]

Collision traits

J'ai viré les méthodes intersects() et contains() de mes classes de bounding volumes, et à la place déclaré des traits de collision dans le namespace wcore::traits.

Un trait de collision est déclaré comme suit :

```
    template<class VolumeA, class VolumeB>
    struct collision
    {
        static bool intersects(const VolumeA& va, const VolumeB& vb)
        {
            // To make the "intersects" relation symmetric
            return collision<VolumeB,VolumeA>::intersects(vb,va);
        }
        static bool contains(const VolumeA& va, const VolumeB& vb);
    };
```

```
-> intersects(a,b) renvoie true si a intersecte b
-> contains(a,b) renvoie true si a contient b
```

Un truc malin que j'ai fait est le comportement par défaut d'intersects(). Si aucune spécialisation n'existe, alors on essaye de renvoyer le résultat de l'intersection pour la classe de trait avec les arguments template permutés. De fait, si collision est spécialisée pour A et B mais pas pour B et A, alors collision::intersects(b,a) renvoie le même résultat que collision::intersects(a,b). Donc on n'a jamais à se casser les couilles pour définir 2 méthodes intersects() pour chaque paire de bounding volumes. En revanche, contains() n'est pas une relation symétrique, et donc on doit fournir 2 spécialisations. Ca implique que si l'on considère les traits collision et collision alors on n'aura vraisemblablement qu'une seule méthode *définie* entre intersects() et contains() dans les 2 specs respectives. Ca pose donc un problème lors du *linking*, s'il manque une implémentation pour contains(). Mais je préfère ce comportement à celui de renvoyer false par défaut pour contains, ce qui entrainera que le programme va compiler et linker mais dysfonctionner.

Pour la collision d'un *FrustumBox* avec n'importe quel type de "boîte" (AABB, OBB, BoundingRegion), j'ai défini une spécialisation partielle qui repose sur l'idée que le type de la boîte en question possède une méthode get_vertices() qui renvoie un std::array contenant les positions des vertices de la boîte en world space (ou en tout cas dans le même repère que les vertices de la *FrustumBox*).

```
    template<class BoxT>
    struct collision<FrustumBox,BoxT>
    {
        static bool intersects(const FrustumBox& fb, const BoxT& bb);
        static bool contains(const FrustumBox& fb, const BoxT& bb);
    };
    template<class BoxT>
    bool collision<FrustumBox,BoxT>::intersects(const FrustumBox& fb, const BoxT&
       bb)
    {
        const std::array<math::vec3, 8>& verts = bb.get_vertices();
        // ...
    }
```

- Note : Si plus tard je dois retourner un résultat plus complexe qu'un simple booléen (genre une variété d'intersection (intersection manifold)), et que le type de retour dépend des paramètres template (un peu comme mes fonctions ray_collides_X() qui écrivent dans des structures RayCollisionData), alors on peut immaginer un truc du genre :

```cpp
template<typename Args...>
struct collision {};

template<class T>
struct collision<T,T>
{
    static bool intersects(const T& t1, const T& t2) { return false; }
    // ...
};

template<class VolumeA, class VolumeB>
struct collision
{
    static decltype(std::declval<collision<VolumeA,VolumeB>>().intersects(
        std::declval<VolumeA>(),std::declval<VolumeB>()))
    intersects(const VolumeA& va, const VolumeB& vb)
    {
        return collision<VolumeB,VolumeA>::intersects(vb,va);
    }
    // ...
};

// + full specs.
```

Ca sort de vieilles notes dans mon carnet noir (!), j'ai collé ça ici pour référence.

- Note : J'ai un problème moral avec l'idée suivante, mais je la note pour référence. On pourrait réécrire la struct collision, de sorte qu'elle utilise la déduction d'argument template (CTAD), mais au prix de lui rajouter un état : "'cpp template struct collision { // For CTAD collision(const T& t, const U& u): t_(t), u_(u) {}

  bool intersects();

private: const T& t_; const U& u_; };

template<> bool collision::intersects() { return do_some_stuff(t_,u_); } template<> bool collision::intersects() { return do_some_stuff(t_,u_); } template<> bool collision::intersects() { return do_some_stuff(t_,u_); } `Utilisation :`cpp Dick dick1; Dick dick2; Bollock bollock; Circle circle;

```cpp
std::cout << collision(dick1,dick2).intersects() << std::endl;
std::cout << collision(dick1,bollock).intersects() << std::endl;
std::cout << collision(dick2,circle).intersects() << std::endl;
```

"'

[15-01-19]

Mesh instance

Mesh caching

Mes *SurfaceMesh* sont maintenant générées par la nouvelle classe *SurfaceMeshFactory* qui joue un rôle symétrique à *MaterialFactory* mais pour les *Mesh*. *SurfaceMeshFactory* possède essentiellement 3 méthodes : - make_procedural() -> Produit un mesh depuis des calculs CPU. Il peut s'agir d'un mesh hard-codé, comme un simple cube, mais aussi de meshes plus complexes (arbre, rocher). - make_obj() -> Produit un mesh depuis un fichier Wavefront .obj - make_instance() -> Produit un mesh procédural depuis un descripteur stocké dans assets.xml -> Utile pour référencer facilement un mesh procédural

Les 3 méthodes tentent de rechercher l'objet à créer dans un cache avant d'en générer un nouveau. * Pour make_obj() et make_instance() c'est facile, il suffit de hasher respectivement le nom de fichier .obj ou le nom de l'instance. Les pointeurs vers les *SurfaceMesh* des .obj et des instances sont dans la même map, associés aux fameux hash. * Pour make_procedural() c'est un peu plus compliqué. 2 mesh de même type générés avec des paramètres différents sont a priori des mesh différents. J'ai donc généralisé la méthode de regroupement des paramètres de génération de *TreeGenerator* et *RockGenerator*. Les structures d'initialisation de générateurs telles que *TreeProps* et *RockProps* héritent d'une interface commune *MeshDescriptor* qui déclare une méthode virtuelle pure : - parse_xml() -> Pour parser un noeud *Generator* dans un fichier XML J'utilise de plus la macro MAKE_HASHABLE de wtypes.h qui permet de générer automatiquement une structure de hashing dans le namespace std pour tout type de struct *MeshDescriptor*. Donc quand je dois créer un nouvel objet procédural, je regarde d'abord le hash de sa structure de description, je le combine au hash de son type avec la macro HCOMBINE_, et je cherche le hash combiné dans un cache réservé aux mesh procéduraux. Je ne crée un nouveau mesh que si la recherche à échoué, auquel cas je mets en cache le nouveau mesh.

[X][fixed] ___BUUUUUUGS___ -> J'ai commenté tout ce qui touche à la mise en cache. -> Commencer par refactor *Model* pour utiliser des shared_ptr pour mesh et material. -> **NE PAS OUBLIER** que la méthode Chunk::load_geometry() va copier le mesh de *CHAQUE* modèle du chunk, qu'il soit mis en cache ou non. De plus, un mesh mis en cache verra son buffer offset et son nelements modifiés *PLUSIEURS FOIS*. -> Problèmes quand on a 2 références au même mesh dans 2 chunks. Comme chaque chunk possède son set de VBOs/IBOs, le mesh possède un couple buffer offset / nelements propre au dernier VBO qui l'a enregistré, et ce couple sera invalide pour référencer le mesh dans le VBO de l'autre chunk. -> Abandonner l'idée de VBOs/IBOs par chunk et centraliser leur gestion dans un système spécialisé ?

[22-01-19]

J'ai implémenté une machine à état pour la gestion du contrôle de la caméra. Un des modes permet de faire un travelling depuis des keyframes enregistrées depuis la cam freefly.

Le nouveau *GameSystem* du nom de *CameraController* repose sur la gestion de plusieurs états internes *CameraState*. *CameraController* est une machine à état qui répond aux événements user selon son état courant, à travers le polymorphisme de *CameraState*. J'ai pour l'instant définit deux états : *CameraStateFreefly* et *CameraStateTrackingShot* (*CameraStateCircleAround* en préparation). Le premier gère la caméra en mode freefly, c'est le mode d'intéraction avec la cam que je traîne depuis le début. Le deuxième permet d'interpoler un mouvement de caméra depuis un ensemble de keyframes (position et orientation de la cam) enregistrées depuis le premier mode. Depuis le mode freefly, la touche "I", permet d'enregistrer une keyframe. Une fois que l'on a accumulé suffisamment de keyframes, on peut générer un interpolateur au moyen de la touche "K", puis changer d'état avec la touche "C" pour parcourir un chemin interpolé en position et en orientation depuis les keyframes enregistrées. Au lieu d'enregistrer les keyframes une par une, on peut en tapant sur "J" démarrer un (fake) enregistrement de la position de la cam. Toutes les 10 frames, si la position ou l'orientation ont suffisamment changé depuis la dernière keyframe, alors une nouvelle keyframe est générée. Taper sur "J" ou "K" arrête l'enregistrement. Le résultat en mode *CameraStateTrackingShot* est assez "shaky" et pas terrible, de plus, changer d'orientation sans changer de position induira un intervalle de taille nulle pour le paramètre (comme l'implémentation courante (de merde) fixe l'intervalle au prorata de la distance des positions de la cam d'une kf à l'autre) et donc le mouvement reproduit ressemblera à un sursaut de souris.

L'interpolation des positions se fait avec une *CSpline* (cardinale). L'interpolation des quats ne peut se faire au moyen d'une *CSpline*, et une lerp basique (+ normalisation) ne fonctionne pas non plus à tous les coups (produit un tour complet de la caméra quand l'assiette change de signe entre 2 keyframes). Pour interpoler 2 quats, on peut utiliser une Slerp (spherical lerp), j'ai pompé l'algo dans l'article wiki [1], la slerp fonctionne impeccablement et contourne les singularités de la lerp comme on s'y attend. L'algo d'origine vient de Shoemake [2]. Cette solution n'est pas toujours la meilleure, basiquement on a 3 caractéristiques que l'on peut souhaiter pour le résultat de l'interpolation, et 3 méthodes possibles qui ne garantissent que 2 de ces caractéristiques parmi les 3 (voir [3]) :

```
1                    commutative   constant velocity   torque-minimal
```

quaternion slerp N Y Y quaternion nlerp Y N Y log-quaternion lerp Y Y N

-> [3] recommande une nlerp (lerp + normalize) en toutes circonstances, c'était mon premier réflexe ici mais ça n'empêchait pas la cam d'effectuer un tour complet d'elle-même quand l'assiette changeait de signe. Et c'est ça qui m'a motivé à chercher une autre solution. En réalité, ce problème vient d'ailleurs. S^3 est un revêtement double de SO(3) (à une rotation R de SO(3) correspondent les quaternions q ET -q), donc indépendamment de l'interpolation, il existe toujours 2 chemins de rotations entre un quat q1 et un quat q2 : un chemin court et un chemin long. Le chemin long produit un "tour complet". Pour forcer l'utilisation du chemin court, il faut inverser le signe d'un des 2 quats si le produit scalaire q1.q2 est négatif. math::slerp() de quaternion.h fait ça, ce qui rend la méthode robuste aux changements de signe de l'assiette. Mais en réalité, je pourrais tout aussi bien implémenter une nlerp qui réalise le même test (ce serait d'ailleurs plus rapide). Oh, well…

Sources :

```
[1] https://en.wikipedia.org/wiki/Slerp
[2] http://run.usc.edu/cs520-s15/assign2/p245-shoemake.pdf
[3] http://number-none.com/product/Understanding%20Slerp
,%20Then%20Not%20Using%20It/
```

[23-01-19]

[TODO][X] Ecrire *EntityFactory* qui permet de construire une entité depuis une description (XML).

```
[X] Pour chaque type de composant, il faut enregistrer une factory method auprès
   de _EntityFactory_, et l'associer avec un nom de composant.
```

[X] Mesh caching [X] Un seul gros VBO pour tous les mesh instances d'un niveau. [X] Mesh pre-pass (dans SceneLoader::load_global() ?) pour générer à l'avance tous les mesh instances et les mesh importés, et les mettre en cache. Les mesh purement procéduraux restent générés à la volée. -> SceneLoader::preload_instances() [X] Les modèles ne possèdent plus nécessairement leurs meshes. Donc utiliser un shared pointer en interne. [X] Les chunks ne doivent plus charger la géométrie des instances. Les instances chargent leur géométrie lors du level loading, dans un gros VBO (de la *Scene* ?). -> Les entités possédant un composant *WCModel* ne peuvent qu'utiliser des instances/imports obj, et donc leurs meshes seront déjà chargées dans ce VBO.

[08-02-19]

J'étais un peu occupé à coder, il semblerait.

hash litteral

On peut maintenant utiliser un string litteral pour hasher une string compile-time :

```
typedef unsigned long long hash_t;
// compile-time hash
extern constexpr hash_t H_(const char* str)
{
    return detail::hash_one(str[0], str + 1, detail::basis);
}
// string literal expression
constexpr hash_t operator "" _h(const char* internstr, size_t)
{
    return H_(internstr);
}
```

```
    std::cout << "hello" << " " << "hello"_h << " " << H_("hello") << std::endl;
```

```
    hello 11831194018420276491 11831194018420276491
```

L'utilitaire internstr a été modifié pour reconnaître le literal.

SoundSystem

On a du son 3D dans le moteur. La classe *SoundSystem* propose une interface simple pour jouer des effets (fx) ou une musique d'ambiance (bgm). Ce *GameSystem* utilise FMOD en sous-main, derrière une implémentation opaque. Deux classes de sons émergent: - les fx sont localisés dans l'espace et leurs caractéristiques sonores dépendent de la position d'un auditeur virtuel (listener), recalé avec la caméra à chaque frame. En temps réel, le moteur de son ajuste l'atténuation et le pitch (effet Doppler) des émetteurs de sons en fonction de la position et de la vitesse du listener. Un effet un peu custom consistant en un filtre passe-bas dont le cutoff est contrôlé par la distance de l'émetteur à l'auditeur simule un effet de propagation. A l'origine je voulais un système physics based, j'ai même chié un simulateur matlab ('Desktop/matlab/atmosphereSoundAttenuation/atten.m') pour calculer la réponse acoustique de l'air sur toute une gamme de fréquences en fonction de la température, de l'humidité et de la pression atmosphérique (voir [3] et [4] pour un modèle mathématique). Puis j'ai compris que j'allais trop loin. - les bgm sont des sons 2D, qui bouclent par défaut.

Le volume (toujours donné en dB côté user, avec 0dB le niveau maximal), les distances min et max pour le falloff (fx) et l'option de jouer le son en boucle sont configurables pour chaque son dans sounds.xml. Le moteur va parser ce fichier et extraire une map de descripteurs pour pouvoir instancier les sons in-game depuis un simple hash name.

Chaque son est joué sur un Channel instancié pour lui-seul par le moteur. Les ressources sont mises en cache. Chaque Channel est une machine à état, ceci permet une gestion beaucoup plus aisée à plus haut niveau, chaque Channel "sait ce qu'il doit faire" selon son état (INITIALIZING, LOADING, PREPARING, PLAYING, STOPPING, STOPPED) lors de l'appel de sa fonction update. L'état d'initialisation (INITIALIZING) est pour l'instant vide, mais pourra accueillir les méthodes de randomisation du pitch/volume etc. Il conduit à l'état de chargement (LOADING) où l'on va chercher la ressource en cache ou la charger depuis un fichier si on ne l'y trouve pas. L'état suivant (PREPARING) initialise le Channel et prépare FMOD à jouer le son dès la frame d'après. Si la préparation échoue, l'état suivant est STOPPED, si elle réussit c'est PLAYING. Un Channel dans l'état PLAYING met à jour ses paramètres en temps réel, et persévère dans cet état tant que le son n'est pas terminé (ou joué en boucle), où que le flag should_stop n'a pas été activé, auquel cas l'état suivant est STOPPING. Dans cet état, un fadeout rapide est exécuté si le son n'était pas terminé, le son est donc arrêté proprement et le channel mis dans l'état STOPPED. Un Channel dans cet état sera supprimé automatiquement à la frame d'après. Les 3 premiers états sont exécutés dans la même frame. Comme j'utilise un switch sur l'état dans SoundSystem::SoundEngineImpl::Channel::update(), j'utilise le *fallthrough statement* pour signaler au compilo qu'on ne break pas et que tout va bien :

```
    switch(state)
    {
        case State::INITIALIZING:
            // Any randomization/adjustment of pitch/volume... goes here
            state = State::LOADING;
            [[fallthrough]];

        case State::LOADING:
        {
            // Load sound if not already loaded
            auto it = impl.sounds.find(sound_id);
            if(it == impl.sounds.end())
                load_sound();
            state = State::PREPARING;
            [[fallthrough]];
        }

        case State::PREPARING:
        {
            state = prepare_play() ? State::PLAYING : State::STOPPED;
```

```
21              return;
            }
23
            // ***
25      }
```

3 Channel groups sont créés par le moteur, ainsi, on peut régler indépendamment le volume des fx et des bgm, et l'on dispose un plus d'un réglage master dans le GUI de l'éditeur (encore en construction).

InitializerSystem

Un peu comme les *GameSystem*, les *InitializerSystem* sont enregistrés dans un *GameSystemCaintainer*. Ce sont les premiers systèmes à produire des données que les *GameSystem* utilisent lors de la phase d'initialisation. Ils peuvent par exemple désérialiser et sérialiser des données afin de rendre persistants certains réglages.

EditorTweaksInitializer

EditorTweaksInitializer est un *InitializerSystem* qui fonctionne un peu sur le principe du singleton CONFIG. Il possède une *ValueMap* qui peut lire/écrire des couples clé/valeur dans le fichier edtweaks.xml et propose un accès aux valeurs via des hashs. Ce système assure la persistance de certains réglages dans l'éditeur (SSAO, post-processing…).

Game object factory

Plusieurs classes factories ont été codées pour alléger le code de *SceneLoader*. J'ai décidé qu'il incomberait à un unique *GameSystem* de type *GameObjectFactory* de créer les objets du jeu. Cela comprend pour l'instant les *Model* et les *WEntity*. *GameObjectFactory* possède deux autres factories pour fabriquer les modèles statiques d'une part, et les entités d'autre part : *ModelFactory* et *EntityFactory*. *ModelFactory* regroupe deux autre factories : *MaterialFactory* et *SurfaceMeshFactory* décrites précédemment.

EntityFactory peut générer des entités depuis un *blueprint*, qui est une description XML du contenu en composants d'une entité. Le fichier entity.xml est parsé au démarrage pour en extraire les blueprints. Les composants sont créés in-situ au moyen d'une factory method qui doit être enregistrée préalablement via EntityFactory::register_component_factory(). Le premier argument est le hash du nom du noeud décrivant un type de composant donné, et le second est une fonction qui ajoute et initialise un composant du type correct depuis une description XML.

GameObjectFactory |___ *ModelFactory* | |___ *MaterialFactory* | |___ *SurfaceMeshFactory* |___ *EntityFactory*

Mesh caching & instances

Le mesh caching a été rétabli pour tous les meshes non procéduraux.

Basiquement, *SurfaceMeshFactory* distingue deux types de meshes : - Les mesh procéduraux : Un mesh procédural est toujoujours généré à la volée au chargement d'un chunk. Il est local à un chunk et est enregistré dans le vertex buffer du chunk. - Les instances Une instance est un objet global de la scène (dans le VBO de la scène), connu à l'avance au chargement de la map. Une pré-passe de *SceneLoader* détermine quels sont les instances qui seront utilisées dans le niveau et charge et met en cache celles-ci. Une instance est déclarée via un descripteur XML dans assets.xml. Ce qui est un peu finaud, c'est que n'importe quel mesh généré procéduralement peut être déclaré en instance tant qu'un descripteur XML existe dans assets.xml. Celà ne concerne pas que les meshes obtenus par chargement d'un fichier.

Le même genre de distinction opère sur les modèles. Un *Model* n'est autre que l'aggrégation d'un *Material*, d'un *Mesh* et d'une *Transformation*. Un modèle utilisant un mesh procédural possède réellement celui-ci, puisque le mesh n'est partagé avec aucun système. Un modèle utilisant une instance de mesh partage en revanche l'instance avec *SurfaceMeshFactory* puisque c'est ce système qui gère le cache. Un modèle déclaré dans assets.xml et pour

lequel à la fois le mesh et le material sont décrits dans assets.xml est une “instance de modèle”. De tels modèles sont générés par *ModelFactory* depuis un simple hash. - L'API supporte l'ajout d'instances de modèles à un chunk non finalisé en run-time. C'est grâce à celà que l'application “maze” fonctionne. - Les entités qui embarquent un composant *WCModel* ne peuvent utiliser que des instances de modèles. C'est *GameObjectFactory* qui enregistre la factory method pour ces composants, puisqu'elle utilise sa *ModelFactory* pour générer les instances de modèles pour l'instant simplement aggrégés dans ceux-ci.

Application maze

Appli test pour me forcer à dev un peu l'API. Un labyrinthe est généré par un algo recursive backtracker depuis une seed dans une structure basée sur un tableau 2D, et la géométrie du niveau est générée en utilisant l'API. Le niveau n'est pas chargé avec LoadStart() mais avec LoadLevel(). Un chunk est chargé mais sa géométrie n'est pas uploadée. Les murs sont instanciés en fonction de la configuration de chaque cellule du tableau, des lumières et des théières (!) sont disposées aléatoirement dans les culs-de-sac. Un espace central au chunk est aussi laissé vide pour ajouter un arbre au centre du labyrinthe. Alors seulement le chunk est uploadé.

Erosion

L'érosion est réparée. L'étape de stitching a été modifiée pour recaler aussi la hauteur des bords. L'érosion par gouttelettes a été modifiée pour éroder le moins possible les bords de chunks, ce qui permet d'éviter de gros artéfacts de raccord. L'érosion est diminuée radialement et linéairement sur toute une bordure intérieure à chaque chunk.

Sources :

```
[1] https://www.youtube.com/watch?v=M8Bd7uHH4Yg
[2] https://katyscode.wordpress.com/2012/10/
05/cutting-your-teeth-on-fmod-part-1-build-environment
-initialization-and-playing-sounds/
[3] http://www.sengpielaudio.com/calculator-air.htm
[4] http://www.sengpielaudio.com/AirdampingFormula.htm
```

[12-02-19]

Distance-enabled parallax mapping

Un nouveau tweak de la pipeline permet de régler la distance minimale d'un objet à la caméra au dessus de laquelle le parallax mapping est désactivé (default to normal mapping). Bonne optimisation.

[17-02-19]

zipios linking

J'ai galéré 3h à essayer de build du foutu code de test pour la lib zipios que je compte utiliser pour gérer les packs de ressources (archives) du jeu :

```
#include "vendor/zipios/zipfile.hpp"

int main()
{
    zipios::ZipFile zipfile("plop.zip");
    zipios::FileEntry::pointer_t entry(zipfile.getEntry("my/resource/file.xml"));
```

```
    zipios::FileCollection::stream_pointer_t
        in_stream(zipfile.getInputStream("my/resource/file.xml"));

    return 0;
}
```

J'avais proprement déclaré la lib en static linking et ajouté son nom dans target_link_libraries() dans le CMakeLists.txt, mais j'aboutissais invariablement à des erreurs de linking "undefined reference". Alors que je faisais tout bien… De plus, une compilation à la main avec clang fonctionnait sans problème. Donc il devait bien y avoir quelque chose dans mon cmake file qui foutait le bordel.

Il s'avère que c'était la ligne suivante :

```
    add_definitions(-D_GLIBCXX_USE_CXX11_ABI=0)
```

Que j'avais ajoutée pour une raison non encore élucidée (comme je n'ai rien documenté à ce sujet). Eh bien zipios dépend explicitement de l'ABI c++11, comme le montre la commande :

```
>> nm -C --defined-only -g ../lib/libzipios.a > libzipios_entries.txt
```

Cette commande liste tous les symboles exportés par la lib (avec les options spécialement pour du code C++). En particulier, on y trouve :

```
0000000000000342 T zipios::ZipFile::ZipFile(std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char> > const&, long, long)
```

Donc on a bien une dépendance à la classe string du constructeur de ZipFile via l'ABI c++11.

Commenter cette ligne rend le build fonctionnel.

En espérant que ça ne me mordra pas le cul plus tard. Pourquoi diable désactivais-je l'ABI C++11 ?!

[21-02-19]

Archives

Le nouveau singleton *FileSystem* d'alias FILESYSTEM regroupe les fonctions d'entrée-sortie. Il est utilisé conjointement au sous-système de gestion des chemins d'accès de CONFIG. *FileSystem* permet de récupérer des std::istream depuis des fichiers en physique, mais aussi depuis des fichiers contenus dans une archive, au moyen de la lib Zipios. Deux fonctions get_file_as_stream() à 2 arguments implémentent ces deux modes d'accès.

Une fonction "smart" get_file_as_stream() qui prend 3 arguments cherche d'abord à obtenir un stream depuis une archive, et le cas échéant depuis un fichier physique. Chaque archive doit inclure un manifeste du nom de MANIFEST.xml à la racine :

```
<?xml version="1.0" encoding="utf-8"?>
<archive>
    <vpath name="model"   value="models/"/>
    <vpath name="texture" value="textures/"/>
</archive>
```

Ce fichier déclare des noeuds *vpath* qui représentent des overrides sur les folder nodes de config.xml. Ainsi il est possible d'avoir une hiérarchie interne dans les archives qui émule une hiérarchie physique de dossiers, par exemple ici, le dossier virtuel models/ à la racine de l'archive est une alternative au dossier physique res/models/ défini par le folder node "model". A l'ouverture d'une archive ce fichier est parsé et la fonction get_file_as_stream(3) à 3 arguments peut se servir de ces informations pour dispatcher correctement vers les fonctions get_file_as_stream(2) à 2 arguments. L'avantage de cette méthode est que l'ajout de nouveaux assets peut se faire dans des dossiers lors d'une phase de test, et ces assets sont ensuite packés dans une archive pour plus d'efficacité (cache friendliness) lorsqu'ils sont suffisamment matures.

```
// After CONFIG.init()
FILESYSTEM.open_archive(fs::path("../res/some_archive.zip"), "some_archive"_h);

auto stream = FILESYSTEM.get_file_as_stream("anvil.mtl", "model"_h,
    "some_archive"_h);
std::string content{std::istreambuf_iterator<char>(*stream),
                    std::istreambuf_iterator<char>()};
std::cout << content << std::endl;

FILESYSTEM.close_archive("some_archive"_h);
```

Une archive par défaut du nom de "pack0.zip" est chargée par le moteur. Cette archive contiendra à terme les assets propres au moteur, mais je m'en sers en ce moment pour mes assets test. Un peu plus tard, il conviendra de définir dans assets.xml dans quelle archive se trouve chaque asset. La fonction wcore::UseResourceArchive() de l'API permet de charger une archive côté user. Cette fonction peut être appelée avant wcore::Init() car elle ne dépend que des données de config disponibles dès la construction d'un objet wcore::Engine.

Tous les systèmes qui réalisaient eux-même les accès io doivent être modifiés pour prendre des streams en entrée. De fait, l'origine des données est opaque de leur point de vue. Pour l'instant, les modèles et les textures sont gérés par le système d'archives. Les fichiers xml restent dans des dossiers pour que je puisse les modifier facilement, même si techniquement *XMLParser* est d'ores et déjà capable de lire des streams. Je modifierai peut être get_file_as_stream(3) pour chercher les fichiers dans les dossiers physiques en priorité, ce qui devrait améliorer le workflow. -> C'est fait. Le *SoundSystem* devra à terme fonctionner sur ce mode, mais la principale emmerde est que je n'ai aucune foutue idée de comment **streamer** un son depuis une archive. FMOD propose bien une fonction pour lire un son depuis la mémoire, ce qui convient sans problème pour les sons de courte durée. Mais les sons longs doivent être streamés et FMOD ne supporte pas std::istream comme entrée.

Better ObjLoader

ObjLoader peut maintenant lire les normales dans un fichier .obj. Elles étaient jusque-là générées par le moteur via l'implémentation de *TriangularMesh*. Si les normales doivent être lues, un *FaceMesh* est généré à la place, les normales sont initialisées depuis les données obj, mais peuvent toutefois être adoucies via l'une des fonctions de smoothing disponibles :

```
        <Mesh name=".interior01" type="obj">
            <Location>interior01.obj</Location>
            <ProcessUV>false</ProcessUV>
            <ProcessNormals>true</ProcessNormals>
            <SmoothNormals>heaviside</SmoothNormals>
            <Centered>false</Centered>
        </Mesh>
```

Le noeud *SmoothNormals* peut prendre les valeurs suivantes : - none - max - heaviside - linear - compress_linear - compress_quadratic

Idéalement j'aimerais avoir un système plus flexible capable de détecter et calculer uniquement les normales manquantes, sans nécessité de dupliquer chaque vertex comme l'implique l'utilisation de *FaceMesh*. Mais ça me demande de refonder complètement les meshs.

[24-02-19]

Cubemap & Skybox

La nouvelle classe *Cubemap* vient abstraire les cubemaps d'OpenGL. Une cubemap est générée depuis un descripteur de type *CubemapDescriptor*, lui-même parsé depuis assets.xml :

```
<Assets>
    <Cubemaps>
        <CubemapTexture name="skybox01">
            <XMinus>sky01_left.png</XMinus>
            <XPlus>sky01_right.png</XPlus>
            <YMinus>sky01_bottom.png</YMinus>
            <YPlus>sky01_top.png</YPlus>
            <ZMinus>sky01_back.png</ZMinus>
            <ZPlus>sky01_front.png</ZPlus>
        </CubemapTexture>
    </Cubemaps>
    <!-- ... -->
</Assets>
```

Ici on définit une texture cubemap depuis 6 fichiers image, un par face. Le descripteur est produit par *MaterialFactory* au chargement du moteur.

Je passe les détails concernant la génération de la texture cubemap, car c'est très semblable à la génération d'une texture2D de base. Voir cubemap.cpp.

La nouvelle classe *SkyBox* utilise en sous-main un *mesh3P* et une *Cubemap* afin de définir un modèle affichable par le *ForwardRenderer*. Le *GameObjectFactory* peut produire un objet *SkyBox* depuis un hashname de cubemap déclaré dans assets.xml. La *Scene* peut optionnellement charger une *SkyBox* via Scene::set_skybox(1), avec comme seul argument un shared_ptr vers une skybox. Le *SceneLoader* s'occupe de celà automatiquement lors du parsing des globales de la scène.

Une map peut déclarer une skybox dans les globales en précisant uniquement le nom d'asset à utiliser pour la cubemap :

```
<Scene>
    <Skybox disable="false">
        <CubemapTexture name="skybox01"/>
    </Skybox>
    <!-- ... -->
</Scene>
```

L'attribut optionnel disable permet de désactiver la skybox rapidement depuis le fichier xml de la map.

Drawing

Une *SkyBox* possède son propre *BufferUnit* et son propre *VertexArray*. Comme c'est un objet unique dans une scène, j'ai voulu faire simple. Je l'ai munie d'une fonction Skybox::draw() pour dessiner la géométrie. Note : j'utilise des Vertex3P car seule la position des vertices est nécessaire pour sampler une cubemap, pas besoin de spécifier de coordonnées UV.

La skybox est un simple cube aligné avec les axes principaux de la map, qui suit la caméra. La partie translationnelle de la view matrix de la cam est annulée, de sorte que la position de la skybox ne soit pas affectée par la position de la caméra dans le repère monde. Elle est toujours centrée sur la cam.

Un shader spécialisé de *ForwardRenderer* (skybox.vert, skybox.frag) est utilisé pour l'affichage. *ForwardRenderer* dessine la skybox en dernier, de sorte qu'un maximum de fragments échoue au depth test. Pour que celà soit possible, il faut se démerder dans le vertex shader pour que la composante en z de la position des vertices soit toujours égale à 1.0 après le perspective divide (profondeur max en coordonnées NDC), et utiliser GL_LEQUAL au lieu de GL_LESS comme fonction de depth test (voir [1]). De fait, un fragment de la skybox ne sera rendu que si aucun fragment n'est plus proche, elle est donc effectivement à l'infini du point de vue de la cam.

skybox.vert :

```
#version 400 core

```

```
layout (location = 0) in vec3 in_position;
out vec3 v3_texCoord;
uniform mat4 m4_view_projection;

void main()
{
    v3_texCoord = in_position;
    vec4 pos = m4_view_projection * vec4(in_position, 1.0);
    // z component always 1 after perspective divide
    gl_Position = pos.xyww;
}
```

skybox.frag :

```
#version 400 core

out vec4 FragColor;
in vec3 v3_texCoord;
uniform samplerCube skyboxTex;

void main()
{
    FragColor = texture(skyboxTex, v3_texCoord);
}
```

Merdique (TODO)

[X] Les textures sont toutes inversées, celà semble provenir du *PngLoader*. J'ai merdiquement inversé les textures de la skybox à la main pour m'en convaincre et laissé ça en état. Il faut régler ce problème rapidement. -> J'inversais volontairement l'ordre des lignes dans *PixelBuffer*, surement pour rendre l'affichage correct du fait que j'inversais aussi la coordonnée V des cubes texturés… [X] Lorsqu'il y a du fog en pleine nuit, le bord de certains objets (branches d'arbre, bord de map) est entouré d'un halo clair à la couleur de la skybox. Je ne sais pas si ce problème persistera quand la skybox sera adaptative (changera de teinte en fonction de l'heure de la journée). -> C'est dû à l'effet bloom et à la FXAA ! Le problème est largement circonscrit en inversant l'ordre forward pass / bloom pass dans la pipeline. La FXAA cause le problème en premier lieu en faisant baver la skybox sur quelques px à l'intérieur des objets, et l'effet bloom élargit le halo ainsi créé. Pour réellement régler le problème, il faudrait générer le fog avant le sampling de la FXAA. Pour l'instant, je me contente d'inverser l'ordre forward pass / bloom pass ce qui implique que l'effet bloom ne pourra pas s'appliquer aux objets rendus en forward. [X] Le mesh de la skybox est sous-optimal, chaque vertex est répété 4 fois, je ne tire pas partie de l'IBO.

Sources :

```
[1] https://learnopengl.com/Advanced-OpenGL/Cubemaps
```

[25-02-19]

A propos de SIMD

J'entretenais quelque velléité de rendre mes classes de maths compatibles SIMD. Mais il y a plusieurs raisons qui entraîneront que je ne le ferai probablement pas. Déjà, le meilleur moyen d'utiliser SIMD est de travailler avec des types compiler-dependent (personne ne se tape de l'assembleur pour une lib math SIMD). Ensuite, je lis que l'efficacité n'est pas nécessairement au rendez-vous lorsque l'on utilise SIMD dans un gros game engine, à cause notamment de la contrainte d'allignement sur 16 bytes qui rend le bordel assez peu cache-friendly.

Voici un com assez sagace d'un type en réponse à TheCherno qui vient de produire une vidéo sur comment et pourquoi il utilisera la lib GLM dans son moteur Hazel :

```
I disagree that simd math is absolutely necessary. It is simply not true that simd
   math is way faster in a huge and complex engine. There are a couple reasons for
   that. First, to make full use of simd, data needs to be stored in these
   registers as long as possible, which is often not the case. Second, simd sse
   data needs to be 16 byte aligned which can also decrease performance, e.g.
   because now you have to load more data in the cache than before and if you
   forgot to manually align it, it will be even slower. Yes the actual
   calculations are faster than non-simd, but simd is not the holy grail of
   graphics programming. And now the reason why i tell you this. Last year i
   thought exactly the same, because everyone teaches it. Now, i work with the
   guys who created the cryengine and did farcry, crysis etc. and they are not
   even using simd math. Back when they tried it it was actually slower in their
   engine. Another thing i want to mention is to have the abilitiy to question
   everything. Today, computer graphics is taught exclusively using matrix math.
   Guess what? Cryengine does not use it except in a few cases (Mainly for
   projection). Matrices are slow and have to much unnecessary data. I would argue
   that replacing matrices with something else (we use dual-quats and qtangents)
   it is way faster than using simd math. Have a nice day! :)
```

Batch image conversion

```
>> mogrify -format png ./*.jpg
```

Convertit tous les .jpg du dossier courant en .png.

[27-02-19]

Splat mapping

J'ai un début de multi-texturing qui fonctionne ! Un terrain patch peut déclarer un material alternatif avec un node *MaterialAlt* dans le fichier XML de la map. Pour l'instant un slider du debug GUI permet d'interpoler linéairement et de manière uniforme sur tout le terrain. Il s'agira de charger en sus de celà une texture par chunk (la splat map) afin de déterminer localement le coefficient d'interpolation. Le shader gpass possède une variante **VARIANT_SPLAT** qui est utilisée par le *GeometryRenderer* pour le rendu du terrain uniquement, si tant est que le terrain en question est multi-texturé (TerrainChunk::is_multi_textured() permet de s'en assurer). Cette variante procède à l'interpolation des valeurs issues de deux *sampler groups*. Un sampler group est une struct du shader qui regroupe pour l'instant 6 sampler2D (albedo, AO, metallic, roughness, normal, depth). Le premier sampler group (celui par défaut) s'appèle sg1 et le deuxième sg2. Donc par exemple pour accéder à la texture albedo du premier sampler group, il faut utiliser le nom d'uniform *mt.sg1.albedoTex* au lieu de mt.albedoTex précédemment.

Beaucoup de ruses à tendance "hacky" ont été utilisées à cet effet, le moteur n'était absolument pas pensé pour que celà soit possible :

- MaterialFactory::make_material(xml_node*) a été modifié pour vérifier le nom du noeud utilisé pour déclarer le material. Si le nom est *MaterialAlt* au lieu de simplement *Material*, alors une valeur sampler_group initialisée à 2 au lieu de 1 est passée au *TextureDescriptor* qui servira à générer la texture alternative.
- La classe *Texture* a été modifiée pour tenir compte du sampler_group du *TextureDescriptor* qui sert à l'initialiser. Au départ, une map statique associant un type enum à un hash_t permettait de sélectionner le sampler name à associer à une texture unit pour permettre le bind automatique des textures aux bons samplers dans les shaders (Texture::SAMPLER_NAMES_). Maintenant, une deuxième map (Texture::SAMPLER_NAMES_2_) qui déclare les noms d'uniforms pour sg2 (ex : mt.sg2.albedoTex) est utilisée alternativement à la première si le sampler_group du *TextureDescriptor* vaut 2. Un peu d'arithmétique permet de s'assurer qu'une texture alternative va bind avec le bon numéro de sampler. Les samplers de sg1

possèdent les indices de 0 à 5, ceux de sg2 de 6 à 11. Les fonctions Texture::unit_to_sampler_name() et Texture::get_unit_index() sont modifiées pour renvoyer les sampler names et unit index corrects en fonction du sampler group de la *Texture*.
- La *Scene* possède une méthode Scene::draw_terrains() qui permet de dessiner les terrains. Scene::draw_models() ne se charge plus de l'affichage des terrains, et ça, c'est une bonne chose en soit. *GeometryRenderer* va d'abord rendre les modèles puis ensuite les terrains lors d'une seconde passe. Si un terrain n'est pas multi-texturé, alors c'est le shader gpass de base qui est utilisé, sinon c'est sa variante **VARIANT_SPLAT**.

A terme, le moteur tentera pour chaque chunk de charger une texture du nom de "splat_levelname_x_z.png" en tant que splat map (avec x et z les coordonnées du chunk), par exemple "splat_crystal_0_0.png" sera utilisée par le niveau "crystal" au chunk (0,0). C'est pour l'instant *ModelFactory* qui va générer ce nom dynamiquement à la création d'un terrain, si une texture alternative a bien été définie et générée avec succès. L'existence de cette texture sera une condition nécessaire pour que le splat-mapping soit activé pour le chunk en question. Une autre condition nécessaire sera que les deux materials définissent les même textures.

A noter que l'activation simultanée du parallax mapping et du splat mapping fonctionne sans problème à ceci près que c'est foutrement lent. -> Ue amélioration possible serait de charger les textures qui devraient l'être en niveau de gris, plutôt que de forcer une conversion RGB pour ensuite ne lire que la composante R dans les shaders comme je le fais pour le moment. Ca devrait libérer pas mal de ressources.

[27-02-19]

Splat mapping

Le splat mapping fonctionne. La classe *Texture* a été augmentée pour pouvoir charger des textures simples depuis un stream. De telles textures sont chargées avec des paramètres par défaut non configurables pour l'instant. *MaterialFactory* peut créer de telles textures simples depuis un stream. ModelFactory::make_terrain_patch(2) vérifie la présence d'un fichier splat map pour le chunk qui doit être chargé via la nouvelle fonction FileSystem::file_exists(3), et si un tel fichier est trouvé, alors la splat map est chargée sous forme de texture simple via *MaterialFactory*, et les ("la" pour l'instant) textures alternatives sont générées et ajoutées au *TerrainChunk*.

Au lieu du noeud *MaterialAlt*, un *TerrainPatch* de fichier map XML peut maintenant définir un noeud *Splat* dans lequel les différentes textures sont déclarées dans l'ordre des sampler groups :

```
<Terrain chunkSize="33" latticeScale="1.0" textureScale="0.25">
    <TerrainPatch origin="(0,0)" size="(5,5)" height="0">
        <!-- ... -->
        <Splat>
            <Material name="beachSand"/>
            <Material name="sandstone"/>
        </Splat>
        <!-- ... -->
    </TerrainPatch>
</Terrain>
```

Ici, le material par défaut est "beachSand" qui sera lié au sampler group 1, et le material "sandstone" sera lié au sampler group 2. Pour l'instant seuls 2 sampler groups sont pris en charge, une future tâche consistera à augmenter ce système pour en gérer jusqu'à 3-4. Noter qu'il est toujours possible de définir un simple noeud *Material* comme auparavant, quand le terrain n'est pas multi-texturé.

Les coordonnées sur la splat map sont calculées dans le vertex shader de gpass dans sa variante **VARIANT_SPLAT** depuis les composantes x et z de la position des vertices dans l'espace modèle, avec un scaling selon la taille du chunk (qui détermine la taille du terrain chunk). Ces coordonnées nommées *landscape coordinates* sont exportées vers les shader stages suivants. Le fragment shader se sert de ces coordonnées pour sampler la splat map et déterminer l'interpolation locale entre les différents sampler groups.

Le nom de la splatmap à charger est produit dynamiquement par *SceneLoader* :

```
splat_[level_name]_[chunk_x]_[chunk_z].png
```

[BUG] Octree non robuste

Lorsque le volume de l'octree de la géométrie statique est suffisamment grand, ce qui peut se produire après avoir parcouru une grande partie de la map, en revenant verse la position initiale de la caméra on peut constater que certaines cellules ne subdivisent plus. -> Profondeur max atteinte ?

Réparation de internstr

Dans la configuration où sur une même ligne j'avais plusieurs string litterals et un hash string litteral, internstr matchait bêtement tout ce qui se trouvait entre le premier et le dernier guillemet, conduisant à une erreur de parsing XML au chargement de debug_intern_strings.xml. J'ai effectué la modification suivante dans les regex :

```
// Avant
static std::regex hash_str_tag("H_\\(\"(.+?)\"\\)");
static std::regex hash_str_literal_tag("\"(.+?)\"_h");
// Après
static std::regex hash_str_tag("H_\\(\"([a-zA-Z0-9_\\.]+?)\"\\)");
static std::regex hash_str_literal_tag("\"([a-zA-Z0-9_\\.]+?)\"_h");
```

De fait, on ne match que les caractères alphanumériques, les underscores et les points, plus les guillemets…

[01-03-19]

Material Editor

Une grosse optimisation à faire serait de pré-compiler toutes les texture maps (Albedo, Roughness, Normal, Metallic, AO et Depth) dans 3 images png 4 canaux :

```
[   Image1   ]  [     Image2     ]  [    Image3      ]
[[R][G][B][A]]  [[R][G][B]  [A]]  [[R]  [G] [B]   [A]]
  Albedo           Normal  Depth  Metal AO  Rough  ?
```

Les sampler groups ne comporteront plus que 3 samplers au lieu de 6 et on divisera vraisemblablement les accès texture par 2.

Mais générer de telles images nécessite un outil, sans quoi ce serait une galère sans nom. Donc il faut que je bosse sur un material editor. On peut imaginer les features suivants :

[X] Génération assistée de la normal map depuis la depth map [X] Par filtrage (Sobel, Scharr…) [X] Possibilité de spécifier des valeurs uniformes au lieu d'images [X] Compilation des textures [ ] Export des materials en XML [X] Visualisation d'objets texturés sous wcore [X] Avec choix des sources de lumière [/] Et choix des objets (cube, plane, sphere, .obj) [/] Opérations de base sur chaque texture map -> Invert, Bias, Curve… [ ] Export sous différentes résolutions

Et un peu de fluff quand j'aurai le temps : [X] Drag & drop de fichiers image [X] Batch mode

Peut-être que pour se simplifier la vie -je sens que je vais regretter le début de cette phrase- et étendre le nombre de formats pris en charge en entrée, on pourrait utiliser une grosse lib image loader genre Assimp.

Le programme serait en standalone (tool) et utiliserait une interface graphique solide (Qt ?).

[05-03-19] Material Editor

J'ai un bon début d'éditeur (target materialeditor) fait avec Qt. Le namespace utilisé est *medit* Le programme utilise *Config* pour se localiser et déterminer les chemins d'accès vers le dossier de travail (res/textures/work/) et le dossier d'export (res/testures/). La fenêtre principale est une classe *MainWindow* qui hérite de QMainWindow. Qt utilise un système de layouts pour positionner les contrôles de manière robuste. Noter qu'il faut remplir les layouts de widgets avant de les attacher à la fenêtre ou à un layout parent avec addLayout().

Dans le panneau de gauche, on trouve une liste QListView pour afficher les textures sur lesquelles on travaille, avec au dessus un input QLineInput et un bouton QPushButton pour ajouter une nouvelle texture (le nom doit être alphanumérique avec éventuellement des underscores), et en dessous un explorateur QTreeView qui permet de se déplacer dans le dossier de travail (dont on suppose qu'il contient un dossier par texture regroupant toutes les texture maps). Il est possible de renommer une texture en double-cliquant sur son nom dans la liste, ou en tapant F2 avec son nom sélectionné, depuis le menu contextuel de la liste, ou encore en cliquant sur le bouton "Rename" de la toolbar. Il est aussi possible de supprimer la texture courante en pressant Suppr, depuis le menu contextuel, ou en cliquant sur le bouton "Delete" de la toolbar. La liste est classée par ordre lexical dynamiquement grâce à un modèle proxy.

Le panneau de droite contient pour l'instant uniquement des contrôles customs *DropLabel* hérités de QLabel, dans lesquels on peut glisser déposer des images, soit depuis l'explorateur du programme, soit depuis l'explorateur de l'OS. Une texture map peut être désinitialisée (clear) depuis le menu contextuel des *DropLabel*.

Une barre d'outils QToolBar regroupe les contrôles les plus utilisés : la sérialisation du projet, l'édition de textures (rename, clear, delete) et la compilation. Les icônes sont généreusement produites par Jess !

Le workflow consiste à créer une nouvelle texture, puis à initialiser les champs qui doivent l'être (les images, mais aussi plus tard les grandeurs uniformes), puis à compiler cette dernière dans 3 images composites (qui plus tard seront concaténées dans une archive). De telles textures composites seront nommées **Blocks**.

Le paradigme de programmation suivi est Modèle/Vue, afin de rester cohérent avec la façon dont fonctionne Qt. La classe *EditorModel* représente les données de l'application et implémente les algos effectifs pour les traîter. *MainWindow* intéragit avec cette classe dans ses slots. *EditorModel* possède une map de descripteurs vers les textures en cours d'édition. Ces descripteurs sont des structures *TextureEntry* qui possèdent en outre des chemins d'accès vers les images sources. Ces descripteurs sont ordonnés par hash du nom de texture.

Utilisation d'un modèle proxy pour le tri d'une liste

Qt propose des widgets qui opèrent une séparation modèle/vue. QListView est l'un d'entre eux (son alternative plus simple est QListWidget de mémoire). Une QListView est construite avec un modèle par défaut mais ce modèle peut être remplacé par une implémentation custom si besoin. Le modèle a besoin d'une source de données pour fonctionner (en l'occurrence une QStringList dans mon cas). Si l'on veut trier dynamiquement les données d'une QListView une possibilité est d'utiliser deux modèles. Le premier est un *modèle source* qui intéragit avec les données directement, et le deuxième est un proxy qui va remapper les indices du modèle source de sorte à réordonner les entrées de la liste dans la vue. Les indices de la vue sont donc ceux du modèle proxy. Ainsi on n'opère jamais sur les données directement pour trier leur représentation :

```
    QListView* listview;
    TexListModel* texlist_model_;
    QSortFilterProxyModel* texlist_sort_proxy_model_;
    QStringList texlist_;
    // ...
    // Custom model uses string list as data
    texlist_model_->setStringList(texlist_);
    // Proxy model for sorting the texture list view when a new item is added
    texlist_sort_proxy_model_->setDynamicSortFilter(true);
    texlist_sort_proxy_model_->setFilterCaseSensitivity(Qt::CaseInsensitive);
    texlist_sort_proxy_model_->setFilterKeyColumn(0);
    texlist_sort_proxy_model_->setSourceModel(texlist_model_);
```

```
13      listview->setModel(texlist_sort_proxy_model_);
```

Voir EditorModel::setup_list_model() dans editor_model.cpp. Il suffit ensuite d'appeler la fonction sort() du proxy à chaque insertion dans la liste :

```
1  QModelIndex EditorModel::add_texture(const QString& name)
   {
3      // Append texture name to list view data and sort
       QModelIndex index = texlist_model_->append(name);
5      texlist_sort_proxy_model_->sort(0);

7      // Add texture descriptor
       texture_descriptors_.insert(std::pair(H_(name.toUtf8().constData()),
           TextureEntry()));
9
       // index is a source index and needs to be remapped to proxy sorted index
11     return texlist_sort_proxy_model_->mapFromSource(index);
   }
```

Noter que les indices fournis par le modèle source doivent être remappés via le modèle proxy pour correspondre aux indices de la vue. L'opération inverse (texlist_sort_proxy_model_->mapToSource()) est nécessaire quand on veut accéder aux données du modèle source depuis un indice de la vue.

Gestion de la sélection dans une QListView

Le mécanisme de sélection de Qt est assez complexe car il doit englober des représentations et des cas d'utilisation très différents. En gros, il y a une séparation entre la notion d'indice courant et de sélection. Cette séparation est ténue dans le cas d'une QListView qui n'autorise pas de sélection multiple. Mon modèle source est custom. J'utilise ma classe *TexListModel* héritée de QStringListModel. Elle définit une fonction append() et un opérateur de stream qui se chargent d'ajouter une entrée dans la QStringList sous-jacente. append() retourne un indice source de type QModelIndex correspondant à la position de la donnée nouvellement insérée dans le modèle source. Un QModelIndex est non scalaire car une QListView peut gérer des entrées à plusieurs colonnes. On utilise ses membres row() et column() pour récupérer la ligne et la colonne. La ligne suivante extraite de EditorModel::delete_current_texture() supprime une ligne entière du modèle source depuis l'indice courant :

```
       texlist_model_->removeRow(texlist_sort_proxy_model_
2          ->mapToSource(tex_list->currentIndex()).row());
```

La sélection elle-même évolue en fonction des intéractions user avec la QListView (click, flèches haut/bas…). Une QListView possède un modèle qui gère la sélection (selection model). Ce modèle est accessible via QListView::selectionModel(). Le modèle de sélection émet un signal selectionChanged() quand la sélection vient de changer, il suffit de connecter ce signal à un slot pour intercepter le changement de sélection :

```
       QObject::connect(tex_list_->selectionModel(),
2                       SIGNAL(selectionChanged(QItemSelection,QItemSelection)),
                        this,
4                       SLOT(handle_texlist_selection_changed(QItemSelection)));
```

Voir mainwindow.cpp. Le slot handle_texlist_selection_changed() de *MainWindow* vérifie que la nouvelle sélection est non vide, procède à la sauvegarde de la texture précédemment sélectionnée, récupère l'indice vers la nouvelle sélection, demande à l'*EditorModel* de changer de texture courante puis enfin met à jour la vue (charge les images de la texture sélectionnée dans les *DropLabel* etc.).

Edition custom

Parfois Qt a recours à des mécaniqmes compliqués pour résoudre des problèmes en apparence simple du point de vue user. La customisation du comportement à l'édition d'une entrée de QListView est un exemple dont ma tension

artérielle porte encore la trace. Comme tous les noms de texture ne sont pas permis (seulement des caractères alphanumériques et des underscores), je dois valider le nom d'une texture à l'insertion, mais aussi au renommage. Donc je dois d'une manière ou d'une autre intercepter l'événement de commit des données dans la liste. Le signal edit() de QListView est généré lorsqu'une intéraction user produit une transition d'état du widget vers un état "editing". Lorsque celà se produit, un "editor widget" est spawné par Qt à l'emplacement de l'édition. Par défaut pour une entrée textuelle dans une QListView il s'agit d'un QLineEdit. En fait, Qt utilise une factory pour produire ces editor widgets à la volée, et ces widgets peuvent assumer à peut près n'importe quelle implémentation sous-jacente. Lorsque l'utilisateur termine l'édition, plusieurs fonctions d'un délégué interne à la liste de type QItemDelegate sont appelées, dont setModelData() qui doit récupérer le contenu du widget editor auquel il n'a accès qu'à travers un type de base QWidget, et copier ce contenu dans le modèle pointé par un index en argument. Et c'est ici que ma classe *TexlistDelegate* intervient.

TexlistDelegate hérite de QItemDelegate et une instance de cette classe remplace le delegate de base dans la QListView. *TexlistDelegate* surcharge setModelData() de sorte à faire intervenir un foncteur de validation du nom. Les données ne sont commit dans le modèle que si le nom est valide selon le foncteur. *TexlistDelegate* demande en outre un pointeur vers le modèle *EditorModel*. Pour installer un tel delegate dans la QListView, je fais ceci dans le constructeur de *MainWindow* :

```
    tex_list_ = new QListView();
    tex_list_delegate_ = new TexlistDelegate();
    // ...
    tex_list_delegate_->set_editor_model(editor_model_);
    tex_list_delegate_->set_item_name_validator(&validate_texture_name);
    tex_list_->setItemDelegate(tex_list_delegate_); // For editing purposes
```

Et voici un extrait de TexlistDelegate::setModelData() :

```
void TexlistDelegate::setModelData(QWidget* editor, QAbstractItemModel* model,
    const QModelIndex& index) const
{
    QLineEdit* line_editor = qobject_cast<QLineEdit*>(editor);
    QString old_name = index.model()->data(index, Qt::EditRole).toString();
    QString new_name = line_editor->text();

    // Check that the name has indeed changed
    if(!old_name.compare(new_name)) return;

    // Validate name and commit
    if(item_name_validator_(new_name))
    {
        hash_t new_hname = H_(new_name.toUtf8().constData());
        if(!editor_model_->has_entry(new_hname))
        {
            editor_model_->rename_texture(old_name, new_name);
            model->setData(index, QVariant::fromValue(new_name));
        }
    }
}
```

Voir texlist_delegate.cpp. Donc on downcast l'editor widget vers un QLineEdit, on récupère les valeurs de l'ancienne entrée et de la nouvelle entrée (pour l'instant dans le champ texte du QLineEdit), on valide la nouvelle entrée grâce au foncteur de validation, on vérifie que le nouveau nom n'existe pas déjà dans la liste, et si ça passe on modifie les données dans le modèle de l'application et dans la liste. EditorModel::rename_texture() va simplement créer une nouvelle entrée sous le nouveau nom, y copier le descripteur de l'ancienne entrée et supprimer cette dernière.

Compilation ad hoc des textures

La fonction EditorModel::compile(const QString& texname) récupère les données pixel des images sources d'un descripteur pointé par le nom de texture en argument. Ces données sont ensuite combinées dans 3 images, les **blocks**. Le type QImage est utilisé plutôt que QPixmap car il permet l'accès bas niveau dont j'ai besoin. Au lieu de générer de telles QImage depuis une conversion des QPixmap contenues dans mes *DropLabel*, ce qui supposerait un couplage trop important avec la vue, je me contente de les générer ab initio depuis les chemins d'accès du descripteur.

Voici comment je génère le block 0 pour référence :

```
    QImage block0(entry.width, entry.height, QImage::Format_RGBA8888);

    QImage** texmaps = new QImage*[NTEXMAPS];
    for(int ii=0; ii<NTEXMAPS; ++ii)
    {
        if(entry.has_map[ii])
            texmaps[ii] = new QImage(entry.paths[ii]);
        else
            texmaps[ii] = nullptr;
    }

    // Block 0
    for(int xx=0; xx<entry.width; ++xx)
    {
        for(int yy=0; yy<entry.width; ++yy)
        {
            QRgb albedo   = texmaps[0] ? texmaps[0]->pixel(xx,yy) : qRgb(0,0,0);
            int roughness = texmaps[1] ? qRed(texmaps[1]->pixel(xx,yy)) : 0;

            QRgb out_color = qRgba(qRed(albedo), qGreen(albedo), qBlue(albedo),
                roughness);
            block0.setPixel(xx, yy, out_color);
        }
    }

    QString block0_name = texname + "_block0.png";
    block0.save(output_folder_.filePath(block0_name));

    delete[] texmaps;
```

Menus contextuels

Un menu contextuel est instancié lors d'un rclick sur un widget. Plusieurs mécanismes sous Qt permettent leur élaboration. J'ai choisi une méthode assez simple (mais pas nécessairement scalable). Déjà, on crée un slot qui produit le menu, par exemple pour le menu contextuel de la liste de textures :

```
void MainWindow::handle_texlist_context_menu(const QPoint& pos)
{
    // Map widget coords to global coords
    // QListView uses QAbstractScrollArea, so we need to get its viewport coords
    QPoint globalPos = tex_list_->viewport()->mapToGlobal(pos);

    QMenu context_menu;
    context_menu.addAction("&Rename", this, SLOT(handle_rename_current_texture()));
    context_menu.addAction("&Delete", this, SLOT(handle_delete_current_texture()));

```

```
    context_menu.exec(globalPos);
}
```

On récupère la position globale du rclick pour pouvoir faire spawner le menu au bon endroit. Pour la plupart des widgets, il suffit de faire :

```
    any_widget->mapToGlobal(pos);
```

Mais dans le cas d'une QListView qui peut être scrollée, il faut en revanche convertir les coordonnées du viewport. Ensuite on génère un QMenu et on lui ajoute des QActions au moyen de la fonction addAction(). Les & marquent un raccourci Alt+. (comme Alt+S pour save ici). La lettre apparaîtra soulignée dans le mot. Si l'on doit mapper ces actions vers des slots déjà existants alors c'est très simple, il suffit de préciser ces slots en argument. Ensuite le menu est spawné via exec() ou popup().

Barres d'outils

Une toolbar fonctionne un peu sur le même principe qu'un menu. Elle est ajoutée à la fenêtre via la fonction addToolBar("toolbar_name") et des actions lui sont ajoutées. Ces actions peuvent prendre une icône en premier argument (on verra après d'où elles viennent), puis un texte et un slot. Des séparateurs peuvent être ajoutés via addSeparator() ou insertSeparator(). Voici un extrait de ma fonction de création de toolbar qui est appelée dans le constructeur de *MainWindow* :

```
void MainWindow::create_toolbars()
{
    toolbar_ = addToolBar("Texture");
    toolbar_->addAction(QIcon(":/res/icons/save.png"), "&Save",
                        this, SLOT(handle_serialize()));
    toolbar_->addAction(QIcon(":/res/icons/save_all.png"), "Save &All",
                        this, SLOT(handle_serialize_all()));

    toolbar_->addSeparator();

    toolbar_->addAction(QIcon(":/res/icons/rename.png"), "&Rename",
                        this, SLOT(handle_rename_current_texture()));
    // ...
}
```

Gestion des ressources

Les fameuses icônes sont des fichiers png localisés dans hosts/materialeditor/res/icons. Toutes ces images sont déclarées dans le fichier resources.qrc dans un format xml-like :

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>res/icons/rename.png</file>
    <file>res/icons/clear.png</file>
    <!-- ... -->
</qresource>
</RCC>
```

Un tel fichier peut être passé au compilateur de ressources de Qt nommé RCC et le pack résultant linké avec l'exécutable. Sous CMake cette opération peut être lancée automatiquement lors du build :

```
    set(CMAKE_AUTORCC ON)
    qt5_add_resources(RCC_SOURCES resources.qrc)
    # ...
    add_executable(materialeditor ${materialeditor_SRCS} ${RCC_SOURCES})
```

Dans le programme, les chemins d'accès vers de tels ressources commencent par ":/", c'est ce que je fais avec les icônes :

```
    QIcon(":/res/icons/save.png");
```

Noter qu'il est possible de définir des alias dans le .qrc :

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file alias="rename.png">res/icons/rename-icon.png</file>
    <!-- ... -->
</qresource>
</RCC>
```

```
    QIcon(":/rename.png");
```

Installation d'un event filter

Ma liste QListView doit pouvoir répondre à l'appui sur la touche Suppr pour supprimer la texture courante. Le moyen le plus simple que j'ai trouvé est de surcharger la fonction eventFilter() dans la classe *MainWindow* et d'y opérer le dispatch en fonction de l'objet et de l'événement concernés :

```
bool MainWindow::eventFilter(QObject* object, QEvent* event)
{
    if (object == tex_list_ && event->type() == QEvent::KeyPress)
    {
        QKeyEvent* ke = static_cast<QKeyEvent*>(event);
        if(ke->key() == Qt::Key_Delete)
        {
            handle_delete_current_texture();
            return true;
        }
        return false;
    }
    else
        return false;
}

// Puis dans le constructeur
{
    // ...
    tex_list_->installEventFilter(this);
    // ...
}
```

Gestion du drag and drop

// TODO

[08-03-19]

Batch convert images

J'avais besoin d'inverser les couleurs d'un ensemble d'icones noires, j'ai utilisé la ligne suivante pour lancer une opération batch :

```
>> for file in *.png; do convert $file -channel RGB -negate w_$file; done
```

[09-03-19]

Stretch bitch

J'ai une classe de base *TexMapControl* qui définit les contrôles de base pour la modification des champs d'une texture map. Je spécialise cette classe par héritage pour chaque type de texmap afin de rajouter des contrôles spécifiques (édition de la valeur uniforme…). Les contrôles additionnels sont rangés dans une QFrame avec un layout QFormLayout, qui place les contrôles en face de labels. J'ai donc créé une classe spécialisée pour l'albédo *AlbedoControl*, avec comme seul contrôle additionnel un *ColorPickerLabel*, qui dérive de QLabel, mais réagit au click, ouvre un color picker dialog et change sa couleur de fond quand une nouvelle couleur est sélectionnée. Tout se passe bien. Je crée une classe spécialisée pour la roughness *RoughnessControl*, et je rajoute comme contrôle additionnel un QLineEdit (maintenant c'est un QDoubleSpinBox beaucoup plus pratique, mais le principe reste le même), et là, surprise, le comportement de changement de taille du *DropLabel* correspondant est modifié, celui-ci s'élargit de manière prioritaire par rapport aux autres quand la fenêtre est redimensionnée. 3h de galère pour biter ce qui m'arrive.

En fait c'est simple, il faut fixer les stretch factors pour chaque élément du layout contenant les *TexMapControl*. Parce que Qt modifie ces derniers dynamiquement en fonction du contenu. Ajouter un label n'a aucun effet de ce point de vue, c'est pour ça que *AlbedoControl* fonctionnait, mais un QLineEdit ou autre champ d'entrée va modifier le stretch factor.

```
    for(int ii=0; ii<texmap_controls_.size(); ++ii)
    {
        layout_main_panel->addWidget(texmap_controls_[ii],ii/3,ii%3);
        layout_main_panel->setRowStretch(ii/3, 1); // So that all texmap controls
            will stretch the same way
        layout_main_panel->setColumnStretch(ii%3, 1);
    }
```

Noter que j'ai organisé mes contrôles dans un layout QGridLayout, on a donc 2 types de facteurs de stretch à modifier, un pour les lignes et un pour les colonnes. Avec un box layout c'est la fonction setStretch() tout court qu'il faut utiliser.

Tant qu'on en est à parler de trucs qui s'étirent, j'ai aussi découvert que pour éviter d'avoir des contrôles qui flottent au milieu de la frame quand on les veut gentiment packés en haut, il faut rajouter un "stretch" après ceux-ci dans le layout qui les contient via :

```
layout->addStretch();
```

Ceci va rajouter une zone extensible qui sera étendue prioritairement en cas de redimensionnement de la fenêtre.

```
Exemple pour un QHBoxLayout :
[[A][B]]  -> [[  A  ][  B  ]] sans stretch
[[A][B]s] -> [[A][B]ssssssss] avec stretch
```

Locale

QCoreApplication et ses dérivés ont un comportement particulier sous les Unix :

```
On Unix/Linux Qt is configured to use the system locale settings by default. This
    can cause a conflict when using POSIX functions, for instance, when converting
    between data types such as floats and strings, since the notation may differ
    between locales. To get around this problem, call the POSIX function
    setlocale(LC_NUMERIC,"C") right after initializing QApplication or
```

```
    QCoreApplication to reset the locale that is used for number formatting to
    "C"-locale.
```

Ici en France, le séparateur décimal est une virgule. Donc tous mes std::to_string formattent les nombres avec une virgule au lieu d'un point comme dans la locale classique (anglaise) du C/C++, ce qui entraînait des erreurs de parsing dans *EditorModel* à l'écriture des données flottantes. J'ai rectifié le tir en réinitialisant la locale après la déclaration de QApplication dans le main():

```
#include <locale>

int main(int argc, char *argv[])
{
    // ...
    QApplication a(argc, argv);
    std::locale::global(std::locale("C"));
    // ...
}
```

C'est dingue le nombre de choses que Qt me fait dans le dos…

Par ailleurs, le formattage des champs d'édition dépend de la locale fixée par Qt, donc un QDoubleSpinBox utilisera des virgules chez-moi. J'imagine qu'on peut rétablir la locale anglaise globalement, mais pour modifier simplement le séparateur décimal pour un controle donné je fais la chose suivante :

```
    QLocale qlocale(QLocale::C);
    qlocale.setNumberOptions(QLocale::RejectGroupSeparator);
    roughness_edit_->setLocale(qlocale);
```

Cela rejette également la virgule comme séparateur de groupes à l'anglaise.

Petite étoile

Je m'étais pété les bollocks à implémenter le comportement d'apparition d'une petite étoile quand les contrôles ont été modifiés et que le projet doit être sauvegardé. Il se trouve que Qt gère ce comportement :

```
    setWindowModified(true);  // -> Appeler ça quand qqchose a changé
    setWindowModified(false); // -> ... quand on vient de sauvegarder
```

Ceci suppose qu'on a bien mis un placeholder pour l'étoile dans la string du titre, qu'on va maintenant modifier comme suit :

```
void MainWindow::update_window_title(const QString& project_name)
{
    setWindowTitle(tr("%1 - %2 [*]").arg(tr("WCore Material Editor"))
                                    .arg(project_name.isEmpty() ? tr("[unnamed
                                       project]") : project_name));
}
```

La macro tr() sert à repérer les chaînes localisables. J'ai pris le réflexe de décorer toutes les chaînes visibles par l'user avec tr(), même si je ne vais pas me faire chier la couille à localiser l'éditeur. Les fonctions arg() se chaînent bien, la première formatte le placeholder %1, la deuxième %2 et [*] est le placeholder pour la petite étoile.

En pratique théorique, setWindowModified() est supposé se propager aux ancêtres quand il est fixé à true, mais pas quand il est à false. En pratique pratique, ça ne se propage quand-même pas chez-moi, et dans tout le code extérieur à *MainWindow* c'est sans effet. Donc y a encore du boulot de recherche et d'expérimentation pour avoir un comportement satisfaisant.

[12-03-19]

Le material editor s'appèle maintenant "Waterial" (merci Jess pour le nom).

[BUILD]

Procédure merdique actuelle pour build depuis zéro :

```
- Installer Qt5 (open source) avec l'online installer
    - Choisir le support des versions 5.11 et 5.12
- Ajouter qmake dans le path :
export PATH="/home/ndx/Qt/5.12.1/gcc_64/bin/:$PATH"
find_package() de CMake va appeler qmake pour explorer l'arborescence de Qt.
```

NE PAS FAIRE

```
>> sudo apt-get install qt5-default
```

```
-> Ca va simplement péter l'install de Qt
```

Cloner mon git et compiler :

```
>> git clone https://github.com/ndoxx/wcore.git
>> cd wcore
>> git submodule init
>> git submodule update
```

```
- Copier à la main le dossier des headers "freetype" dans vendor
```

```
>> mkdir build; cd build
>> cmake [-DCLANG6=1] ..
>> make wcore
>> make sandbox
>> make maze
>> make materialeditor
```

```
[ ] Simplifier ce bordel
```

J'avais un gros problème pour build materialeditor chez Jess. Qt ne détectait pas le standard c++17. Au départ ça a déclenché un bug avec cotire : "c++17 enabled in PCH but currently disabled". Désactiver cotire permettait alors de comprendre le problème sous-jacent : plein de messages d'erreurs s'affichaient pour me prévenir que certains éléments de syntaxes faisaient partie du draft c++17.

Dans tous mes CMakeLists.txt (et en particulier dans celui de materialeditor) j'ai remplacé

```
add_definitions(-std=c++17)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17 ")
```

par

```
set(CMAKE_CXX_STANDARD 17)
```

Et là tout a fonctionné. Je suppose que Qt attendait un flag -std=c++1z pour fonctionner où que sais-je. Cette nouvelle notation permet d'être indépendant de la string associée au standard.

Image Processing

J'ai implémenté les algos et les contrôles pour générer des AO maps et des Normal maps depuis des Depth/Height maps. Les algos sont basés sur le travail de Christian Petry (http://cpetry.github.io/NormalMap-Online/). Le gars utilise des shaders pour produire les textures, j'ai simplement traduit son code en C++ et l'ai adapté pour qu'il tourne rapidement sur un CPU et soit compatible avec les types QImage de Qt. J'ai aussi apporté quelques corrections ça-et-là, notamment en renormalisant les noyaux de Sobel-Feldman et Scharr. Le noyau de Scharr que j'utilise est différent du sien, j'ai choisi une version optimale en terme d'estimation de gradient (au sens des moindres carrés). J'ai aussi repris son code pour le filtre blur/sharpen. Le filtre blur est Gaussien séparable 5x5 et sharpen est un *Unsharp Mask*.

Scrollable

Comme mes contrôles sont de plus en plus nombreux j'ai décidé de les regrouper dans une zone scrollable en dessous de chaque texture map. La scrollbar se masque et s'affiche automatiquement en fonction des dimensions de la fenêtre.

Dans le ctor de TexMapControl :

```
    QVBoxLayout* layout = new QVBoxLayout();
    QFrame* additional_controls = new QFrame();
    QScrollArea* scroll_area = new QScrollArea();
    // ...
    scroll_area->setObjectName("TexmapScroll");
    scroll_area->setBackgroundRole(QPalette::Window);
    scroll_area->setFrameShadow(QFrame::Plain);
    scroll_area->setFrameShape(QFrame::NoFrame);
    scroll_area->setWidgetResizable(true);
    scroll_area->setWidget(additional_controls);
    layout->addWidget(scroll_area);
    this->setLayout(layout);
```

Les classes spécialisées n'ont plus qu'à définir un layout pour la QFrame, y poser des widgets et ajouter un stretch après :

```
    QDoubleSpinBox* metallic_edit_ = new QDoubleSpinBox();
    QFormLayout* addc_layout = new QFormLayout();
    addc_layout->addRow(tr("Uniform value:"), metallic_edit_);
    // ...
    additional_controls->setLayout(addc_layout);
    add_stretch(); // calls layout->addStretch();
```

La scrollarea utilisera la palette window par défaut pour son coloriage, par cohérence stylistique j'ai rendu son fond transparent et retapé la scrollbar, car il fallait bien qu'elle soit orange.

Pour styliser la scroll area :

```
    QScrollArea#TexmapScroll
    {
        background: transparent;
    }
    QScrollArea#TexmapScroll > QWidget > QWidget
    {
        background: transparent;
    }
    /*QScrollArea#TexmapScroll > QWidget > QScrollBar
    {
        background: palette(base);
```

```
    }*/
    QScrollBar:vertical
    {
        border: 2px solid grey;
        background: rgb(255,120,26);
        width: 15px;
        margin: 20px 0px 20 0px;
    }
    QScrollBar::handle:vertical
    {
        background: rgb(255,150,26);
        min-height: 20px;
    }
    QScrollBar::add-line:vertical
    {
        border: 2px solid grey;
        background: rgb(255,120,26);
        height: 20px;
        subcontrol-position: bottom;
        subcontrol-origin: margin;
    }
    QScrollBar::sub-line:vertical
    {
        border: 2px solid grey;
        background: rgb(255,120,26);
        height: 20px;
        subcontrol-position: top;
        subcontrol-origin: margin;
    }
    QScrollBar:up-arrow:vertical, QScrollBar::down-arrow:vertical
    {
        border: 2px solid grey;
        width: 3px;
        height: 3px;
        background: white;
    }
```

Icons sur Ubuntu/Gnome

Pour que l'éditeur soit accessible via un raccourci dans ma launchbar, j'ai dessiné une icone que j'ai placée dans /usr/share/pixmaps/ et j'ai créé un fichier waterial.desktop dans ~ /.local/share/applications :

```
[Desktop Entry]
Version=1.0
Name=Waterial
Comment=Waterial is a material editor for the WCore engine.
Exec=/home/ndx/Desktop/WCore/bin/waterial
Path=/home/ndx/Desktop/WCore/bin/
Icon=/usr/share/pixmaps/waterial_128.png
Terminal=false
Type=Application
Categories=Utility;Development;
```

Il suffit ensuite de glisser-déposer le .desktop dans la launchbar. Cette opération doit être automatisée dans la target install en cas de déploiement.

[15-03-19]

Taille du fb

J'essaye d'intégrer le moteur dans un widget pour réaliser la preview de Waterial. Bien entendu, ça ne fonctionne pas, j'ai un fond noir et des cheveux en moins. Une analyse avec apitrace semble montrer que ma scène est rendue correctement dans le LBuffer, mais *le framebuffer par défaut (backbuffer) est de taille 1x1.* Vraisemblablement, quelque chose a foiré lors de la génération du contexte.

Récupérer de manière indirecte la taille du framebuffer par défaut :

```
    GLint dims[4] = {0};
    glGetIntegerv(GL_VIEWPORT, dims);
    GLint fbWidth = dims[2];
    GLint fbHeight = dims[3];
    std::cout << "FB: " << fbWidth << " " << fbHeight << std::endl;
```

Je fais ça avant et après mon rendu de frame dans GLWidget::paintGL(), mais je n'observe pas de taille délirante.

Pour une frame donnée, j'observe l'état d'OpenGL lors du dernier draw call qui correspond à l'affichage à l'écran :

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, NULL)
```

J'ai vérifié qu'un trace de la sandbox montre bien une taille correcte du framebuffer par défaut, ce n'est pas un problème lié à apitrace a priori.

Je crois que je progresse. Le default framebuffer du widget n'est PAS 0 (voir [1]). Dans GLWidget::paintGL() je fais :

```
    std::cout << defaultFramebufferObject() << std::endl;
```

Et ça m'affiche 10. Mon programme tente de bind sur 0, il est logique que ça ne fonctionne pas. Je vais essayer d'installer un hook dans ma lib pour choisir le framebuffer par défaut à bind depuis le code appelant.

MOTHERFUCKER C'était bien ça ! J'ai ajouté une fonction SetDefaultFrameBuffer() à l'API qui va modifier les membres statiques DEFAULT_FRAMEBUFFER de *FrameBuffer* et du wrapper GFX. Au lieu d'appeler glBindFramebuffer(GL_FRAMEBUFFER, 0) le code va lancer glBindFramebuffer(GL_FRAMEBUFFER, DEFAULT_FRAMEBUFFER). En revanche cet appel doit être effectué dans paintGL() pour que cela fonctionne :

```
void GLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    engine_->SetDefaultFrameBuffer(defaultFramebufferObject());
    engine_->Update(16.67/1000.f);
    engine_->RenderFrame();
    engine_->FinishFrame();
}
```

Pour référence, je suis tombé sur un snippet intéressant pour remplir une QImage dans paintGL() :

```
    makeCurrent();
    glPushAttrib(GL_VIEWPORT_BIT);
    glViewport(0, 0, width(), height());
    QOpenGLFramebufferObject fbo(width(), height(),
       QOpenGLFramebufferObject::CombinedDepthStencil);
    fbo.bind();
    glClearColor(1, 1, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Draw stuff
```

```
10
    fbo.release();
12  glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
14  glPopAttrib();
    QImage fboImage(fbo.toImage());
16  QImage image(fboImage.constBits(), fboImage.width(), fboImage.height(),
        QImage::Format_ARGB32);
    image.save("out.png");
```

Comment faire un widget OpenGL

Dans un premier temps, il y a des variables à ajuster au niveau global pour la génération du contexte. Dans le main on fait :

```
1   QSurfaceFormat fmt;
    fmt.setDepthBufferSize(24);
3   fmt.setStencilBufferSize(8);
    fmt.setVersion(4, 0);
5   fmt.setProfile(QSurfaceFormat::CoreProfile);
    QSurfaceFormat::setDefaultFormat(fmt);
```

Juste avant la création de la fenêtre *MainWindow*. Ensuite on crée un objet qui hérite de QOpenGLWidget. Classiquement, on veut aussi hériter en protected d'une des classes QOpenGLFunctions_xx pour avoir accès à l'API :

```
class GLWidget : public QOpenGLWidget, protected QOpenGLFunctions_4_0_Core
2 {
    // ...
4 };
```

Noter que la classe QOpenGLFunctions tout court existe, mais correspond à l'API OpenGLES2. Donc on s'expose à des invalid read si on utilise ça pour du code OpenGL4.x. En pratique, comme mon code utilise GLew et qu'il y a une incompatibilité entre les deux, je ne peux me le permettre, et j'omets simplement cet héritage.

La classe *GLWidget* doit surcharger plusieurs fonctions :

```
protected:
2   void initializeGL() Q_DECL_OVERRIDE;
    void paintGL() Q_DECL_OVERRIDE;
4   void resizeGL(int width, int height) Q_DECL_OVERRIDE;
    void mousePressEvent(QMouseEvent* event) Q_DECL_OVERRIDE;
6   void mouseMoveEvent(QMouseEvent* event) Q_DECL_OVERRIDE;
```

initializeGL() est chargée d'initialiser les fonctions d'OpenGL. Si l'on utilise l'héritage sur QOpenGLFunctions_xx on peut y appeler initializeOpenGLFunctions(). Dans mon cas, c'est GLew qu'il faut initialiser :

```
void GLWidget::initializeGL()
2 {
    // Initialize GLEW
4   glewExperimental = GL_TRUE; // If not set, segfault at glGenVertexArrays()
    if (glewInit() != GLEW_OK) {
6       DLOGF("Failed to initialize GLEW.", "core", Severity::CRIT);
        fatal("Failed to initialize GLEW.");
8   }
    glGetError();   // Mask an unavoidable error caused by GLEW
10
    // ...
```

```
12      engine_->Init(0, nullptr, nullptr, context_);
        engine_->LoadStart();
14  }
```

Noter que j'ai aussi dû créer une nouvelle classe *QtContext* héritant de *AbstractContext*, qui est pour l'instant un gros stub de merde, mais qui doit néanmoins être passée à Engine::Init() afin que le moteur ne crée pas de contexte GLFW par défaut (voir qt_context.h/cpp). Je ne sais pas encore comment je vais m'y prendre pour la remplir, parceque la gestion des evts par Qt est très différente de celle de GLFW qui a inspiré l'interface d'*AbstractContext*.

La fonction paintGL() est appelée à chaque update du widget. C'est là qu'on dessine le stuff. Il est recommandé de toujours la commencer par un glClear() afin de permettre à certains hardwares (en particulier embarqués) de bien déduire l'état du double-buffering, sans quoi on peut s'attendre à de grosses baisses de perfs.

```
    void GLWidget::paintGL()
2   {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4
        engine_->SetDefaultFrameBuffer(defaultFramebufferObject());
6       engine_->Update(16.67/1000.f);
        engine_->RenderFrame();
8       engine_->FinishFrame();
    }
```

La fonction resizeGL(2) est appelée à chaque evt de redimensionnement du widget. Je me contente d'y provoquer un update des propriétés GLB.WIN_W et GLB.WIN_H pour le moment. Ce sont ces variables qui sont utilisées lors de l'appel à glViewport() dans mon code (en tout cas lors du rendu à l'écran).

C'est à peu près tout.

Orientation de départ de la caméra

On savait que c'était de la merde, maintenant on sait aussi pourquoi. Ce connard de GLFW se réserve le droit de retourner des valeurs nulles pour la position du curseur (via glfwGetCursorPos()) :

```
1   Any or all of the position arguments may be NULL. If an error occurs, all non-NULL
        position arguments will be set to zero.
```

(Voir [2]) C'est ce qui se produit lors des 2 premières frames, va comprendre pourquoi. Ca entraîne un evt mouse move sur le chan *input.mouse.locked*, et c'est ça qui baise l'orientation de la caméra de 30° exactement (en pitch et en yaw).

Donc j'ai modifié InputHandler::handle_mouse() pour détecter cette situation et ne pas poster d'événement sur input.mouse.locked quand ça se produit.

A faire

On veut pouvoir faire les choses suivantes depuis l'API : [X] Référencer un modèle donné via un hash [X] Swapper le material d'un modèle référencé [ ] Swapper le mesh d'un modèle référencé. En l'occurrence il me faut pouvoir afficher le material sur : [X] Un plan [X] Un cube [ ] Une sphère [ ] Un .obj quelconque [X] Modifier le mouvement d'un objet référencé, même si c'est complètement hacky dans un premier temps. [X] Activer/Désactiver plusieurs systèmes de rendu de la pipeline. Basiquement, j'aimerais pouvoir faire une passe géométrique, une passe lighting et puis c'est marre. [X] Intéragir dynamiquement avec les sources lumineuses.

Le moteur doit aussi pouvoir : [X] Se passer complètement de terrain dans ses chunks. -> Pour déclarer un terrain patch vide, il faut ajouter l'attribut void="true" dans le node *TerrainPatch*.

Sources :

```
[1] https://forum.qt.io/topic/48816/qopenglcontext-s
-defaultframebufferobject-always-returns-0-in-a-qopenglwidget-subclass
[2] https://www.glfw.org/docs/latest
/group__input.html#ga01d37b6c40133676b9cea60ca1d7c0cc
```

[17-03-19]

Valgrind

Pour tracker l'origine d'une "uninitialized value" :

```
>> valgrind --track-origins=yes ../bin/sandbox -l mv
```

```
==6595== Conditional jump or move depends on uninitialised value(s)
==6595==    at 0x81B7496: ??? (in /usr/lib/nvidia-410/libnvidia-glcore.so.410.104)
[...]
==6595==  Uninitialised value was created by a stack allocation
==6595==    at 0x8314D30: ??? (in /usr/lib/nvidia-410/libnvidia-glcore.so.410.104)
```

Cette erreur se produit dans le *GeometryRenderer* au premier appel à glClear(), même si glClearColor() a été appelé avant. C'est probablement un faux positif, et j'ai marqué cette erreur pour la suppression dans valgrind.supp :

```
{
    <NVidia-driver>
    Memcheck:Cond
    obj:/usr/lib/nvidia-410/libnvidia-glcore.so.410.104
}
```

```
>> valgrind --suppressions=../valgrind.supp ../bin/sandbox -l mv
```

Qt est aussi du genre à générer beaucoup de faux positifs, faudra que je me bricole un fichier de suppression à l'occasion.

Pour générer une suppression facilement (voir [1]), créer une application minimale et lancer :

```
>> valgrind --leak-check=full --show-reachable=yes --error-limit=no
--gen-suppressions=all --log-file=minimalraw.log ./minimal
```

Ca va générer un gros log qui contient toutes les sorties de valgrind, et chaque erreur est accomapgnée d'une suppression. Alors le gawk script tools/parse_valgrind_suppressions.sh permet d'extraire les suppressions et de les sortir dans un fichier de suppression :

```
>> cat ./minimalraw.log | ./parse_valgrind_suppressions.sh > minimal.supp
```

const-correctness

Petit bout de code assez marrant (voir [2]) :

```
struct Type
{
  int _i = 0;
  void modify(Type & t) const
  {
    t._i = 1;
  }
```

```
};

int main()
{
  Type t;
  t.modify(t);
  assert (t._i == 1);
}
```

```
An object changed its own value from within a const-qualified member function!
   Everything is const-correct!
```

Sources :

```
[1] https://wiki.wxwidgets.org/Valgrind_Suppression_File_Howto
[2] https://akrzemi1.wordpress.com/2014/06/02/ref-qualifiers/
```

[21-03-19]

Camera

J'ai corrigé les WTF de la classe caméra. Jusque là, les raisons du fonctionnement correct de cette classe n'étaient pas totalement élucidées (!).

J'ai remarqué des similarités entre les fonctions update() et look_at(). Ces fonctions mettent toutes les deux à jour la view matrix (et la model matrix), mais selon des stratégies différentes. J'ai donc implémenté des politiques d'initialisation en mode enum pour regrouper tout le code d'update dans la fonction update().

```
void Camera::update(float dt)
{
    // * Update frame interval
    dt_ = dt;

    // * Update view matrix according to policy
    if(view_policy_ == ViewPolicy::ANGULAR)
        math::init_view_position_angles(view_, position_, math::vec3(0.0f,
           TORADIANS(yaw_), TORADIANS(pitch_)));
    else if(view_policy_ == ViewPolicy::DIRECTIONAL)
        math::init_look_at(view_, position_, lookat_, vec3(0,1,0));

    // * Extract proper axes
    right_   = vec3(view_.row(0));
    up_      = vec3(view_.row(1));
    forward_ = vec3(view_.row(2));

    // * Update frustum bounding box
    if(update_frustum_)
        frusBox_.update(*this);
}
```

L'ancienne fonction maths::init_rotation_euler() qui calculait une matrice de rotation depuis 3 angles utilisait en réalité des angles de Tait-Bryan dans la convention ZYX. Donc j'ai renommé la fonction en conséquence : init_rotation_tait_bryan(). On lui passe dans l'ordre le roll (Z) le yaw (Y) et le pitch (X). Cette fonction utilisait un produit matriciel en interne, je l'ai remplacée par une dérivation directe des termes de la matrice en me servant de [4], voir d'ailleurs la table [euler_angles_to_matrix.png] dans les figures. En politique ANGULAR (par défaut),

update() appèle math::init_view_position_angles() qui calcule la matrice de rotation depuis les angles, puis translate celle-ci du vecteur position. La matrice de vue renvoyée est l'inverse affine de la matrice modèle (pourra être optimisé). En politique DIRECTIONAL, c'est la fonction math::init_look_at() qui est utilisée pour initialiser la matrice de vue. Les vecteurs right, up et forward sont les colonnes de la matrice modèle et donc les lignes de la matrice de vue. *Camera* ne possède d'ailleurs plus de matrice modèle comme membre.

Noter que forward pointe vers les z négatifs car j'utilise un repère indirect (lefty) à la OpenGL pour les matrices de vue. En revanche dans le repère monde c'est un repère direct qui est utilisé, le calcul de la frustum box doit changer le signe du vecteur forward renvoyé par la caméra pour produire des vertices dans le repère monde. A terme je vais probablement me foutre en direct partout. Voir [3] pour les détails. [1] et [2] m'ont servi à re-re-revérifier le calcul de mes matrices de projection via math::init_orthographic(), math::init_perspective() et math::init_frustum().

Sources :


```
[1] http://www.manpagez.com/man/3/glOrtho/
[2] http://www.manpagez.com/man/3/glFrustum/
[3] https://www.3dgep.com/understanding-the-view-matrix/
[4] https://en.wikipedia.org/wiki/Euler_angles
```


[24-03-19]

LinearPipeline

J'ai implémenté une petite abstraction sympa pour Waterial. J'ai eu un cas fréquent à gérer : une fenêtre de dialogue avec des contrôles et une zone de rendu OpenGL. Les contrôles intéragissent dynamiquement avec le rendu. C'est le cas des "image tweaks" qui permettent d'apporter quelques modifications sommaires à une image source (pour l'instant des ajustements HSV) et d'utiliser l'image modifiée au lieu de l'originale. C'est le cas également de la génération d'AO et normal maps que j'ai entièrement retapé pour la faire avec des shaders (donc la boucle est bouclée, si on se souvient que j'avais traduit des shaders en C++ pour faire la première version).

Ces différents cas de traîtement d'image nécessitent une pipeline minimaliste pour organiser des passes de rendu. C'est là qu'interviennent les classes *ShaderStage* et *LinearPipeline*. Un *ShaderStage* représente une passe de rendu. Il contient un shader program et un FBO. Une *LinearPipeline* chaîne plusieurs *ShaderStage*, de sorte que le color attachment du FBO d'un étage est samplé par l'étage suivant. Le dernier étage n'a pas de FBO initialisé, c'est la pipeline qui choisit le FBO de sortie : soit le FBO par défaut du contexte, soit un FBO membre utilisé pour récupérer l'image de sortie et l'enregistrer dans un fichier png. Le nombre d'étages est une conséquence de l'initialisation. On passe au constructeur un vecteur de paires de QString contenant les chemins d'accès vers les shaders (paire vertex/fragment pour chaque étage). L'initialisation des étages se fait alors automatiquement.

```
NormalGenGLWidget::NormalGenGLWidget(QWidget* parent):
ShaderGenGLWidget({
                    {":/res/shaders/passthrough.vert",
                        ":/res/shaders/gen_normal.frag"},
                    {":/res/shaders/passthrough.vert",
                        ":/res/shaders/blur_h.frag"},
                    {":/res/shaders/passthrough.vert",
                        ":/res/shaders/blur_v.frag"},
                  },
                  parent),
// ...
```

Une telle pipeline est instanciée dans *ShaderGenGLWidget*. Cette dernière classe hérite de QOpenGLWidget et de QOpenGLFunctions_4_0_Core, et définit toutes les fonctionnalités nécessaires à un traîtement multi-étage avec possibilité d'export du rendu. Les classes *TweaksGLWidget*, *AOGenGLWidget* et *NormalGenGLWidget* héritent toutes de cette classe abstraite, et définissent des chaînes de traitement pour les image tweaks, la génération d'AO maps, et la génération de normal maps respectivement. Ces widgets embarquent des membres correspondant aux

uniformes à envoyer à GL, plus des slots pour les updater. Ces trois widgets sont embarqués dans trois dialogues : *TweaksDialog*, *AOGenDialog* et *NormalGenDialog*. Chaque contrôle de chaque dialogue est connecté au slot qui va bien, de sorte qu'une modification de valeur entraîne un update du rendu.

Note : Les dialogues **doivent** être détruits (delete) à leur fermeture et créés (new) avant leur réouverture. Il ne m'a pas été possible de les réinitialiser, basiquement j'avais besoin de changer la taille des FBOs à chaud, et donc de les détruire et recréer, mais un thread de Qt essaye d'intéragir avec ceux-ci, donc à moins d'arriver à foutre un mutex sur le FBO (je ne sais même pas si Qt permet ça), y a juste pas moyen. Ca ne change rien du tout au final, c'est juste beaucoup plus simple.

[26-04-19]

Refactor

Le singleton *GeometryCommon* enregistre toute la géométrie utilitaire (quads…) et de debug et la rend disponible à tous les renderers. Quand on veut dessiner un quad dans un renderer, au lieu se faire chier la couille avec une fonction load_geometry() qui génère un quad dans le *BufferUnit* du renderer et un appel à buffer_unit_.draw(), il suffit de faire :

```
    CGEOM.draw("quad"_h);
```

Les renderers n'ayant plus besoin de *BufferUnit* ont été détemplatés et *Renderer* est maintenant une simple interface. Chaque renderer doit toujours surcharger la virtuelle pure render(), mais cette dernière est maintenant protected. La fonction Render() se charge d'appeler l'implémentation render() si le renderer est activé. Par ailleurs, le profiling a été poussé dans la fonction Render() quand **PROFILE** est défini. Ainsi, RenderPipeline::render() ne fait plus qu'une quinzaine de lignes et se contente d'appeler la fonction Render() de tous les renderers à la suite.

[28-04-19]

SSR: Screen Space Reflections

J'ai implémenté il y a deux semaines un renderer spécialisé pour les réflexions. J'ai choisi une approche screen-space qui semble suffir pour les scènes que je pense avoir à gérer. Peut-être que je devrai implémenter quand-même des cubemaps reflections plus tard, il est assez fréquent dans l'industrie d'utiliser ces deux systèmes conjointement pour produire les réflexions.

J'ai envisagé que l'implémentation d'un algo SSR puisse être plus aisée que prévu après avoir visionné la vidéo de [1]. J'ai commencé par écrire les classes *SSRRenderer* et *SSRBuffer* sur le modèle de *SSAORenderer* et *SSAOBuffer*, puis un shader en m'inspirant de cette source. Je savais ce code très sous-optimal à l'avance, mais il présentait l'avantage d'être très compréhensible visuellement :

- On calcule la position view space du fragment, le gars fait ça avec son position buffer, moi je renconstruis cette position depuis le depth buffer. Le vecteur position obtenu est aussi le vecteur direction allant de l'origine de la caméra au fragment, toujours en view space.

```
1 vec3 fragPos = reconstruct_position(depth, frag_ray, rd.v4_proj_params);
  vec3 fragNormal = normalize(decompress_normal(fNormMetAO.xy));
```

- On normalise ce vecteur et on le reflette en se servant de la normale locale.

```
vec3 reflected = normalize(reflect(fragPos, fragNormal));
```

- Le vecteur obtenu est la direction qui part du fragment jusqu'à la position de l'objet réfléchi (encore à découvrir). Il suffit d'itérer le long de cette direction, jusqu'à ce que la profondeur du fragment atteint (hit point) dépasse celle du depth buffer, ce qui se produit quand le vecteur réfléchi intersecte une surface solide. Là on retourne les coordonnées screen-space du dernier hit point, et on s'en sert pour sampler une

texture du rendu de la frame précédente. “‘c vec3 dir = reflected; vec3 hitCoord = fragPos; for(int ii=0; ii<rd.i_raySteps; ++ii) { // March ray: advance hit point hitCoord += dir; // Project hit point to screen space projectedCoord = rd.m4_projection * vec4(hitCoord, 1.0); projectedCoord.xy /= projectedCoord.w; projectedCoord.xy = projectedCoord.xy * 0.5 + 0.5;

```
// Get linear depth of nearest fragment at hit point from depth map
depth = depth_view_from_tex(depthTex, projectedCoord.xy, rd.v4_proj_params.zw);

dDepth = -hitCoord.z - depth;
if(dDepth > 0.f)
    return hitCoord.xy;

dir *= rd.f_step;
```

}

// …

out_SSR = texture2D(lastFrameTex, hitCoord.xy).rgb;

“‘

Bien entendu on a un beau bestiaire d’artéfactes dégueulasses à gérer, et plusieurs solutions pour mitiger chacun d’entre eux :

- *Banding* Comme on itère avec un pas fixé, la position finale du hit point obtenue par l’algo précédent est systématiquement derrière l’objet, et le bruit de quantification qui en résulte provoque des bandes sur les réflexions. Une solution efficace est de lancer quelques itérations de recherche binaire pour affiner la position du hit point. [1] et [2] proposent une implémentation.
- *Toujours du banding* On peut ajouter un peu de bruit à la position de départ du rayon pour améliorer le résultat davantage (dithering).
- *Faux positifs* Liés au fait que le depth buffer n’a pas d’épaisseur. On peut simuler une épaisseur homogène ou bien se servir d’un *backface depth buffer* obtenu avec un rendu cull front dont la différence avec le front depth buffer est un bon estimateur de l’épaisseur de la géométrie visible. J’avais lu un papier impressionnant utilisant de telles *thickness maps* pour estimer du SSS. [4] mentionne cette possibilité. Mon moteur est capable de produire un backface depth buffer, mais j’ai eu peu de succès avec cette méthode. -> Implémentation réussie, voir le [21-05-19]
- Et bien d’autres.

A ce stade, plutôt que de poursuivre la résolution de problèmes sur du code-jouet, je me suis intéressé à des implémentations plus sérieuses, dont beaucoup semblent reprendre le travail de Morgan McGuire [3]. Notamment deux implémentations intéressantes par Kode80 [4] et [5] en HLSL et Pissang [6] en GLSL (qui d’ailleurs recolle la SSR avec du physically based, et fait du importance sampling sur une GGX Schlick pour simuler les réflexions diffuses des matériaux rugueux).

L’idée assez élégante de McGuire est de faire le raymarching en 2D raster plutôt qu’en 3D. Les relations entre l’avancement du rayon en 3D et sa projection à l’écran sont estimées par le calcul différentiel. Kode80 y ajoute du fading pour les rayons qui approchent le nombre max d’itérations, les côtés de l’écran, ou vont vers la caméra. Une méthode est aussi proposée pour simuler les réflexions diffuses avec un flou Gaussien adaptatif dont la taille du noyau varie en fonction de la rugosité locale.

Mon implémentation reprend pour l’essentiel le travail de Kode80 et Pissang. J’ai aussi une implémentation du flou adaptatif avec un *PingPongBuffer*, mais c’est extrêmement lent en raison du nombre énorme d’accès textures exigé. Et le bénéfice reste très minime sur les scènes que j’ai testées, donc je l’ai désactivé par défaut. Ma SSR est franchement pas dégueu pour du half-res et tourne assez rapidement (0.5ms-1.5ms selon les scenes). J’ai fait pas mal de micro-optim (genre pour forcer des MADs, éviter des divisions…). J’ai toujours un artéfact qui me rend dingue si j’y fait trop attention, mais il ne se produit que près du sol, quand on regarde dans la direction de l’origine en world space (WTF), ce qui ne peut pas se produire dans les scènes que j’envisage, donc au pire… Pour référence, voici quelques observations sur cet artéfact : * Des bandes grises / de texture répétée se forment à distance fixe de la caméra sur un sol réfléchissant. * Les fragments de la zone affectée sont calculés en une seule itération et présentent une discontinuité UV avec leur voisinage. * Ces fragments passent le test d’intersection avec le depth

buffer, ça me fait évidemment penser que ce sont des faux positifs. * L'artéfact disparaît complètement quand on regarde dans la direction des x et z croissants (world space). * La magnitude de l'artéfact ne semble pas dépendre de la position en world space, juste de l'orientation azimuthale de la caméra. * L'artéfact ne dépend pas de la résolution du framebuffer, ni d'ailleurs d'aucun paramètre actionnable du shader.

Sources :


```
[1] http://imanolfotia.com/blog/update/2017/03/11/ScreenSpaceReflections.html
[2] https://gitlab.com/congard/algine/blob/master/src/resources/shaders/
fragment_screenspace_shader.glsl
[3] http://casual-effects.blogspot.com/2014/08/screen-space-ray-tracing.html
[4] http://www.kode80.com/blog/2015/03/11/screen-space-reflections-in-unity-5/
[5] https://github.com/kode80/kode80SSR/blob/master/Assets/Resources
/Shaders/SSR.shader
[6] https://github.com/pissang/claygl-advanced-renderer/blob/master
/src/SSR.glsl
[7] http://roar11.com/2015/07/screen-space-glossy-reflections/
[8] http://bitsquid.blogspot.com/2017/06/reprojecting-reflections_22.html
[9] https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/
[10] https://thomasdeliot.wixsite.com/blog/
single-post/2018/04/26/Small-project-OpenGL-engine-and-PBR-deferred-pipeline
-with-SSRSSAO
```


Framebuffer Peek enfin utile

J'ai beaucoup amélioré cette fonctionnalité du *DebugOverlayRenderer* qui me permet depuis le menu debug d'afficher n'importe quelle texture interne du moteur préalablement enregistrée. Au lieu de simplement afficher cette texture dans une fenêtre ImGui, je fais un rendu avec un shader spécialisé (fbpeek.vert/frag) qui implémente les features suivants : * Tone mapping (on/off) * Filtrage des channels R G et B * Inversion de la couleur * Split screen avec position ajustable du split, pour afficher la valeur du canal alpha en niveaux de gris à droite * Si un unique canal est observé alors on bascule en mode niveau de gris, plus lisible qu'un niveau de couleur primaire. * On peut toutefois afficher la texture en l'état (raw) si on le souhaite. * On peut sauvegarder les textures telles que visionnées dans la fenêtre, dans des images png !

La sauvegarde d'images mérite un commentaire. Basiquement, j'utilise glReadPixels sur le framebuffer ciblé par cette passe de rendu pour populer un buffer, et j'utilise une nouvelle fonction write_png(4) de *PngLoader* (qui ferait bien de changer de nom pour l'occasion) qui utilise la libpng pour encoder dans un fichier png le contenu d'un buffer :

```
    BufferModule render_target_;

    // ... Render ...

    int width = render_target_.get_width();
    int height = render_target_.get_height();
    int img_size = width * height * 4;
    unsigned char* pixels = new unsigned char[img_size];

    render_target_.bind_as_target();
    glPixelStorei(GL_PACK_ALIGNMENT, 1);
    glReadPixels(0, 0, width, height, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8_REV,
        pixels);
    render_target_.unbind_as_target();

    fs::path file_path("path/to/file.png");
    png_loader.write_png(file_path, pixels, width, height);
    delete[] pixels;
```

Noter que j'utilise render_target_.bind_as_target() et non bind_as_source(), parce que c'est le frabuffer que je cherche à bind et non la texture. Peut être que je devrais renommer cette fonction en bind_framebuffer() ? J'avoue m'être planté la première fois, comme quoi c'est pas clair. Le changement de pack alignment à 1 permet grossièrement d'éviter à glReadPixels() d'écrire out of bounds en essayant à tout prix d'aligner des blocks de données sur 4 bytes (par défaut), comme on a réservé une taille exacte pour le buffer (hauteur x largeur x 4 canaux RGBA). Le GL_UNSIGNED_INT_8_8_8_8_REV est supposé être plus rapide (par un certain mec d'un certain forum qui citait un certain mec de NVidia).

Shader line numbers

J'ai donc ce système fantastique qui me permet de gérer un niveau d'include dans mes shaders. Le problème immédiat que celà entraîne, et je l'avais noté à l'époque (voir [05/06-09-18]), c'est que les numéros de lignes renvoyés par le shader error report en cas de problème de compilation, sont décalés de ceux que j'observe dans l'éditeur, du fait que chaque directive include est remplacée par un certain nombre de lignes avant que le source ne soit envoyé au compilo. Je me suis payé une migraine un jour, incapable de retrouver une ligne problématique et j'ai décider d'agir.

Il me suffit de compter le nombre de lignes avant et après parsing dans la fonction compile_shader(3), la différence des deux peut être retranchée aux numéros de ligne renvoyés par le rapport d'erreurs pour obtenir les numéros de lignes corrects, visibles dans mon éditeur. Il reste à parser le rapport d'erreur pour obtenir les numéros des lignes problématiques :

```
1    std::set<int> errlines;

3    // ... Get error report in string logstr

5    // * Find error line numbers
     static std::regex rx_errline("\\d+\\((\\d+)\\)\\s:\\s");
7    std::regex_iterator<std::string::iterator> it(logstr.begin(), logstr.end(),
        rx_errline);
     std::regex_iterator<std::string::iterator> end;
9
     while(it != end)
11   {
         errlines.insert(std::stoi((*it)[1]));
13       ++it;
     }
15
     // ... In compile_shader()
17
     int nlines_raw = std::count(shader_source_raw.begin(),
        shader_source_raw.end(), '\n');
19
     // ... Parse shader code
21
     int nlines_parsed = std::count(shader_source.begin(), shader_source.end(),
        '\n');
23   int line_offset = nlines_parsed - nlines_raw;

25   // ... Further down in error handling section

27   // * Show problematic lines
     std::istringstream source_iss(shader_source);
29   std::string line;
     int nline = 1;
31   while(std::getline(source_iss, line))
     {
33       if(errlines.find(nline++)!=errlines.end())
```

```
        {
35          int actual_line = std::max(0, nline-line_offset-1);
            std::cout << actual_line << " : " << line << std::endl;
37      }
    }
```

Je me sers du pattern

```
    /\d+\((\d+)\)\s:\s/
```

pour matcher les numéros de ligne dans un texte qui ressemble à ça :

```
1 0(376) : error C1503: undefined variable "jj"
  0(376) : error C0000: syntax error, unexpected ')', expecting ',' or ';' at token
     ")"
3 0(282) : error C1110: function "ray_march" has no return statement
  0(392) : error C0000: syntax error, unexpected '.', expecting "::" at token "."
```

Je capture le nombre entre parenthèses qui m'intéresse. Ensuite pour chaque ligne du source, si la ligne cause une erreur, je l'affiche avec son numéro de ligne.

Maintenant c'est putain de limpide, je regrette de ne pas m'être donné la peine plus tôt :

```
  0(378) : error C1503: undefined variable "quequette"
2
  > 179 :             dPQKj = dPQK * stride * quequette;
4 [0.431286][sha] ‡   Shader will not compile: SSR.frag
```

Shader hot swap

J'ai implémenté une fonctionnalité permettant de recharger un shader program à chaud. On peut maintenant éditer les shaders pendant que le moteur tourne !

Un shader qui déclare la directive :

```
#pragma hotswap
```

est enregistré dans une map statique de *Shader* en tant que candidat potentiel au rechargement dynamique. Lorsqu'on presse la touche **F8** l'ensemble des shaders référencés par cette map est rechargé grâce à un appel à la statique Shader::dbg_hotswap().

Le rechargement d'un programme peut échouer sans danger. Pour recharger un programme je procède comme suit : * De nouveaux shader ids sont créés. * Les fichiers sources sont rechargés et compilés. * Si la compilation échoue j'affiche un warning avec le rapport d'erreurs. La fonction de compilation s'occupe déjà du cleanup. Echec du rechargement. A ce stade, c'est comme si rien ne s'était passé du point de vue du moteur. * Je crée un nouveau programme et j'y link les shaders nouvellement compilés. * Si le linking échoue, je libère les ressources et échoue le rechargement. * Si tout s'est bien passé, je libère les ressources openGL de l'ancienne version du programme et je substitue les anciens IDs par les nouveaux. Le rechargement a réussi.

Le rechargement est effectif en début de frame dans RenderPipeline::render(). L'appel à Shader::dbg_hotswap() se fait entre deux glFinish() par précaution.

Always-on-top

Comme il est particulièrement irritant que la fenêtre de l'appli se cache derrière l'éditeur quand je commence à éditer un shader (par exemple), j'ai cherché le moyen de rendre celle-ci flottante (always-on-top). Il suffit d'appeler :

```
1 glfwSetWindowAttrib(window_, GLFW_FLOATING, GLFW_TRUE);
```

Seul problème : cette fonction était introduite en version 3.3, j'avais la 3.1.2… J'ai donc build la dernière version depuis la source (hyper simple) :

```
>> git clone https://github.com/glfw/glfw.git
>> cd glfw
>> nano CMakeLists.txt
```

```
Pour activer le build de la shared lib et désactiver tout le reste
```

```
>> mkdir build;cd build
>> cmake ..
>> make
```

Puis j'ai copié les .so dans WCore/lib et les includes dans WCore/source/vendor/GLFW. Maintenant je link avec cette version, tout se passe bien.

Bref, pour activer le mode always-on-top il faut mettre à true la propriété *root.display.topmost* de config.xml.

[01-05-19] Fête du travail, ça tombe bien, y en a beaucoup

Plan de route pour le moteur d'animation

J'attaque la partie animation. Je me suis fixé sur la possibilité d'animer des modèles 3D relativement low poly pour mon jeu. La difficulté comme à chaque fois que je démarre un gros système est de trouver un angle d'attaque ; comme souvent je commence côté données.

[/] Créer un tool pour convertir des exports Blender vers mes formats propriétaires. * Un type de fichier en entrée * Deux fichiers en sortie : un pour le modèle (vertex data), un pour le squelette (bone hierarchy) et l'ensemble des animations applicables à ce squelette. -> Ainsi on pourra trivialement réutiliser un squelette et un groupe d'animation d'un modèle à un autre semblable. [ ] Augmenter le mesh loader actuel ou en créer un nouveau, pour être en mesure de charger les nouvelles données per-vertex (bone IDs / weights). [ ] Créer un loader pour initialiser une classe *SkelettalAnimation* qui prendra en charge deux structures de données : [ ] Un arbre *BoneHierarchy* qui contient la description hiérarchique d'un squelette, chaque os est à un noeud de l'arbre et possède une matrice d'offsets. [ ] Une hashmap contenant des objets *Animation*, lesquels contiennent les delta-transformations pour chaque os pour chaque keyframe. [ ] Etendre la classe *Model* ou créer une nouvelle classe *AnimatedModel* qui prend en charge des meshes avec le nouveau format de vertex, et un tableau de matrices représentant la pose instantannée du modèle (joint transforms). [ ] Créer un système *AnimationSystem* (hériterait de *GameSystem*) qui associe des modèles animés avec une instance de *SkelettalAnimation*, et met à jour les joint transforms de chaque modèle animé enregistré. [ ] Créer un *Renderer* spécialisé pour les modèles animés. Peut-être que du forward rendering conviendrait mieux.

J'ai démarré une classe template *Tree* pointer-less que je teste en ce moment. Je dispose d'un fichier XML test représentant une hiérarchie d'os, j'ai eu l'idée de traverser le DOM en depth-first et d'initialiser le tableau de noeuds interne à l'arbre de manière linéaire, à mesure que je rencontre un noeud lors de la récursion. Après initialisation, un parcours linéaire du tableau équivaut à un parcours depth-first de l'arbre. En gros j'ai baked un depth-first traversal dans un tableau et j'appèle ça un arbre. Bien sûr, les éléments du tableau, les noeuds, contiennent l'information hiérarchique. Cette méthode m'assure également une utilisation optimale du cache lors du calcul itératif des transformations, en effet, un noeud et son fils seront toujours contigus en mémoire, et comme les transformations d'un noeud donné dépendent toujours de la transformation du parent, les données nécessaires à chaque itération sont calculées à l'itération précédente.

ATTENTION : * Les bone IDs per-vertex sont des entiers, bien penser à utiliser glVertexAttrib*I*Pointer au lieu de glVertexAttribPointer pour bind les IDs dans la structure de données per-vertex.

[04-05-19]

Je bosse sur l'utilitaire *wconvert* qui permet de convertir un modèle animé au format Collada (.dae) en fichiers lisibles par le moteur. Le programme se borne à parcourir un dossier de travail où sont stockés les fichiers d'échange exportés par Blender, et à convertir ces derniers en une paire de fichiers contenant respectivement les données d'animation squelettale (pour l'instant juste la hiérarchie de l'armature) et les données de mesh.

WConvert a des attentes particulières pour les meshes animés en entrée. Un seul mesh doit être présent dans le fichier d'échange, le cas échéant seul le premier mesh déclaré sera parsé. Le noeud de l'armature **doit** s'appeler "Armature".

J'ai codé un importer basé sur Assimp du nom de *AnimatedModelImporter*, lequel remplit une structure *AnimatedModelInfo* avec des données de mesh et une hiérarchie squelettale *Tree*. Un autre importer du nom de *StaticModelImporter* spécialisé pour les meshes statiques remplit une structure *StaticModelInfo*.

Assimp gère automatiquement la triangulation, la génération des normales, tangentes et bi-tangentes. Une des difficultés a été de récupérer les poids et bone IDs qui sont stockées dans les os pour Assimp et d'en faire des données par vertex.

J'ai également deux exporters capables de lire des structures *AnimatedModelInfo* et *StaticModelInfo*, et d'exporter les données d'armature (dans le cas de meshes animés) et de mesh vers des fichiers distincts :

XMLSkeletonExporter

Cette classe exporte une hiérarchie squelettale dans un fichiers XML à l'extension .skel. Un tel fichier tire partie de la hiérarchie du DOM pour représenter la hiérarchie squelettale. Chaque noeud définit en prime une matrice d'offsets en bone space pour le positionnement correct des os :

```
<?xml version="1.0" encoding="UTF-8"?>
<BoneHierarchy name="chest">
    <offset>1.061462 0.000000 0.000000 0.000000 0.000000 1.034307 -0.238556
       0.000000 0.000000 0.238556 1.034307 0.000000 -0.006071 0.537051 0.012778
       1.000000</offset>
    <bone name="Armature_bottom">
        <offset>0.942097 0.000000 0.000000 0.000000 0.000000 0.211730 -0.917996
           0.000000 0.000000 0.917996 0.211730 0.000000 0.005720 0.802042 0.704223
           1.000000</offset>
        <bone name="Armature_top">
            <offset>0.942097 0.000000 -0.000000 0.000000 -0.000000 0.483948
               -0.808296 0.000000 0.000000 0.808296 0.483948 0.000000 0.005720
               -0.405061 0.609134 1.000000</offset>
        </bone>
    </bone>
</BoneHierarchy>
```

BinaryMeshExporter et *WeshLoader*

Cette classe exporte les données de mesh (per-vertex data) dans un fichier binaire .wesh (!), dont le format est pris en charge par la classe WCore *WeshLoader*.

Un wesh file possède la structure suivante :

```
[HEADER]              -> 128 bytes, padded
[size_t vsize]        -> number of vertices
[array of VertexAnim] -> vertex buffer content of size vsize
[size_t isize]        -> number of indices
[array of uint32_t]   -> index buffer content of size isize
```

Le header commence par un magic number 0x48534557 (WESH en ASCII), et définit le numéro de version du format utilisé pour l'export :

```
struct WeshHeader
{
    uint32_t magic;
    uint8_t version_major;
    uint8_t version_minor;
};

#define WESH_HEADER_SIZE 128
typedef union
{
    struct WeshHeader h;
    uint8_t padding[WESH_HEADER_SIZE];
} WeshHeaderWrapper;
```

L'astuce pour forcer une taille de header à 128 octets (padding) consiste à envelopper la structure du header dans une union avec un tableau de 128 octets, et à (dé)sérialiser le wrapper plutôt que le header lui-même. En mode lecture, *WeshLoader* commence par parser le header, et détermine si le wesh file est valide (présence du magic number et numéro de version compatible). L'avantage de ce format est que les vertices sont stockées avec des données entrelacées, telles qu'utilisées par le moteur, ainsi je peux directement sérialiser / désérialiser un couple vertex buffer / index buffer de la façon la plus rapide possible.

La lecture et l'écriture se fait au moyen de streams, rendant ce système compatible avec FILESYSTEM :

```
    // --- WRITE to std::ostream& stream ---
    size_t vsize = vertices.size();
    // Write vertex data size
    stream.write(reinterpret_cast<const char*>(&vsize), sizeof(vsize));
    // Write vertex data
    stream.write(reinterpret_cast<const char*>(&vertices[0]),
        vertices.size()*sizeof(VertexAnim));

    // --- READ from std::istream& stream ---
    size_t vsize;
    // Read vertex data size
    stream.read(reinterpret_cast<char*>(&vsize), sizeof(vsize));
    // Read vertex data
    std::vector<VertexAnim> vertices(vsize);
    stream.read(reinterpret_cast<char*>(&vertices[0]), vsize*sizeof(VertexAnim));
```

En pratique, *WeshLoader* possède des méthodes read() et write() paramétrées par le format de vertex, ainsi tous les formats de vertices sont (dé)sérialisables automagiquement.

En particulier, le format wesh prend en charge les mesh animés, présentant le vertex layout suivant :

```
math::vec3 position_;
math::vec3 normal_;
math::vec3 tangent_;
math::vec2 uv_;
math::vec4 weight_;
math::i32vec4 bone_id_;
```

C'est le format de vertex défini par la structure *VertexAnim* de vertex_format.h.

[05-05-19]

Notes sur Assimp

Je m'étonnais du fait que mes modèles importés comportaient exactement autant d'indices que de vertices. Par ailleurs la liste des indices était simplement la liste des entiers ordonnés. Il se trouve qu'Assimp a recours par défaut à la duplication des vertices lors du post-processing, de telle sorte que seul le vertex buffer est pertinent. Ainsi, en suivant l'écrasante majorité des tutos tels que [1] qui ignorent cette subtilité on se retrouve dans ce cas de figure des plus inefficaces.

Il m'a fallu creuser dans une doc assez mal branlée avant de découvrir les bons flags de post-processing à activer lors de l'import pour pallier ce problème (voir [2] pour une liste complète des flags). Dans assimp_utils.h je déclare les flags utilisés par mes deux importers :

- Convert n-gons to triangles
  - *aiProcess_Triangulate*
- Detect degenerate faces, next flag will ensure they are removed and not simply collapsed
  - *aiProcess_FindDegenerates*
- Split meshes with different primitive types into submeshes. With previous flag, will remove degenerates.
  - *aiProcess_SortByPType*
- Remove/fix zeroed normals / uvs
  - *aiProcess_FindInvalidData*
- Reduce the number of input meshes
  - *aiProcess_OptimizeMeshes*
- Reorder triangles so as to minimize average post-transform vertex cache miss ratio
  - *aiProcess_ImproveCacheLocality*
- Validates indices, bones and animations
  - *aiProcess_ValidateDataStructure*
- Remove parts of input data structure, such as vertex color, to allow for efficient vertex joining
  - *aiProcess_RemoveComponent*
- Allow vertices to be shared by several faces
  - *aiProcess_JoinIdenticalVertices*
- Generate smoothed normals if normals aren't present in input data
  - *aiProcess_GenSmoothNormals*
- Generate tangents and bi-tangents
  - *aiProcess_CalcTangentSpace*
- Flip UVs vertically and adjust bi-tangents -> *aiProcess_FlipUVs*

Si un modèle définit des couleurs per-vertex, ce dont WCore n'a rien à foutre, ces données pourront néanmoins entrainer une duplication de vertex, donc il est important de les filtrer en entrée pour optimiser le vertex joining. Pour ce faire, il faut configurer l'importer avant utilisation :

```
    importer_.SetPropertyInteger(AI_CONFIG_PP_RVC_FLAGS, aiComponent_COLORS);
```

et utiliser le flag de post-process *aiProcess_RemoveComponent*.

D'autre part, la correction de faces dégénérées (faces avec des vertices identiques) avec *aiProcess_FindDegenerates* produit des faces non triangulaires par collapsing dont il faut se débarrasser. A cet effet, on utilise le flag *aiProcess_SortByPType* qui va séparer un mesh possédant des primitives non-homogènes en divers sub-meshes homogènes, parmi lesquels les non-triangulaires seront ignorés grâce à cette configuration :

```
1     importer_.SetPropertyInteger(AI_CONFIG_PP_SBP_REMOVE, aiPrimitiveType_POINT
                                                           | aiPrimitiveType_LINE);
```

Le flag *aiProcess_ImproveCacheLocality* est intéressant, il permet de limiter les cache misses lors du rendu, j'avais déjà lu à ce sujet auparavant. L'algo Tipsify est utilisé (voir [3]).

Sources :

```
[1] https://opengl.developpez.com/tutoriels/ogldev-tutoriel/22-assimp/
[2] http://sir-kimmi.de/assimp/lib_html/postprocess_8h.html
[3] https://gfx.cs.princeton.edu/pubs/Sander_2007_%3ETR/tipsy.pdf
```

[07-05-19]

J'essaye d'avancer la content pipeline binaire. Waterial peut désormais exporter les matériaux dans un format binaire proprio : les Wat files. Je documenterai ce format plus tard. Pour l'instant, il me faut refactor la classe *Texture* qui est vraiment trop merdique, si je veux pouvoir continuer sans peine. En effet, j'aimerais changer la stratégie de mise en cache des textures, centraliser le cache dans la *MaterialFactory*.

Textures : MEGA refactor

J'ai viré les "named textures", artéfact d'un lointain passé. L'objectif que cherchait à accomplir ce feature était de rendre accessibles globalement certaines texture targets internes au moteur. En pratique, seuls les *BufferModule* globaux comme *GBuffer*, *LBuffer* et autres, en profitaient. Mais comme tous ces buffers étaient déjà des objets globaux (singletons), le feature faisait doublon, et je me retrouvais avec soit des accès aux textures nommées, soit des accès aux *BufferModule* globaux, au gré de mon humeur, rendant le code globalement imbitable.

Les singletons posaient d'ailleurs un problème en soi. Toutes les classes suivantes ont dégagé : *GBuffer*, *LBuffer*, *SSAOBuffer*, *SSRBuffer* et *ShadowBuffer*. Tous ces *BufferModule* sont maintenant enregistrés proprement dans une classe statique *GMODULES*, et sont accessibles via la méthode GMODULES::GET(hash_t name) :

```
    GMODULES::GET("gbuffer"_h);
    GMODULES::GET("shadowmap"_h);
    GMODULES::GET("SSAObuffer"_h);
    GMODULES::GET("SSRbuffer"_h);
    GMODULES::GET("bloombuffer"_h);
    GMODULES::GET("lbuffer"_h);
```

Le petit nouveau "bloombuffer" vient d'un refactor de *BloomRenderer* qui utilisait encore un FBO et une texture séparés, alors que *BufferModule* est justement fait pour regrouper un FBO et une texture cible. *BloomRenderer* n'utilise plus que des *BufferModule* en interne.

Pour enregistrer un *BufferModule* accessible globalement (qu'on appèlera GModule par la suite), il suffit de faire :

```
    GMODULES::REGISTER(std::make_unique<BufferModule>(/*args*/));
```

Par ailleurs, les *BufferModule* utilisent des std::unique_ptr en interne, plutôt que des std::shared_ptr.

ATTENTION à bien initialiser des lvalue references avec GMODULES::GET() :

```
    auto& l_buffer = GMODULES::GET("lbuffer"_h);
```

Sinon le FBO interne pourra être détruit, causant un double-free plus loin.

[10-05-19]

Textures : MEGA refactor -> suite et fin

Ré-écriture complète de la classe *Texture*. J'ai viré cette horreur de Texture::TextureInternal qui était mon moyen jusque-là d'assurer le caching des textures (je crois me souvenir que l'idée venait de BennyBox@yt, sympa au départ mais plus du tout adaptée à mon moteur). Toutes les méthodes inutiles ont été supprimées, l'interface est propre, quasi-OpenGL-agnostique et largement commentée, beaucoup de simplifications ont été réalisées dans l'implémentation (meilleure gestion des sampler groups, fonctions helpers…). La mise en cache des textures est réalisée par *MaterialFactory*. La méthode MaterialFactory::cache_cleanup() est utilisée pour supprimer du cache

les textures non partagées (sorte de garbage collector). Cette méthode est appelée par la fonction update() de *GameObjectFactory* à intervalle régulier (toutes les 10s). Et surtout, les textures peuvent être chargées depuis des **WatFiles** !

De nombreux bugs m'ont pourri la vie. Notamment un avec Waterial, très étrange. Lors de la compilation d'une texture, toute la surface du contexte était remplacée par la texture, mais en bleu (blue texture of death)… J'ai mis du temps à me rendre compte que j'appelais glActiveTexture juste avant de bind lors de la génération des textures OpenGL, ce qui vraisemblablement rentrait en concurrence avec un des threads de Qt. Apitrace a encore été de bon secours. En revanche, Valgrind beaucoup moins !

J'ai eu ce bug affreux de Valgrind décrit en [1], qui ne reconnait pas l'instruction assembleur correspondant à l'OpCode RDRAND (génération de nombre aléatoire avec une source d'entropie hardware du CPU). Qt a je pense mis à jour ses libs, et semble maintenant utiliser cette instruction (en tout cas chez-moi) dans une de ses fonctions rand. Quoi qu'il en soit, valgrind échouait à débugger Waterial à cause de ça. J'ai essayé de patcher la source de Valgrind à la main (en suivant le conseil d'un gars, mais sans réellement comprendre ce que je faisais), rien n'y faisait. Le problème semble être résolu après un changement de version de valgrind. Noter qu'on peut compiler Qt avec QT_NO_CPU_FEATURE=rdrnd pour éviter l'utilisation de RDRAND.

Pour référence, voici les typedefs d'OpenGL (voir aussi [2]) :

```
    typedef unsigned int  GLenum;
    typedef unsigned char GLboolean;
    typedef unsigned int  GLbitfield;
    typedef void    GLvoid;
    typedef signed char GLbyte;   /* 1-byte signed */
    typedef short   GLshort;  /* 2-byte signed */
    typedef int   GLint;    /* 4-byte signed */
    typedef unsigned char GLubyte;  /* 1-byte unsigned */
    typedef unsigned short  GLushort; /* 2-byte unsigned */
    typedef unsigned int  GLuint;   /* 4-byte unsigned */
    typedef int   GLsizei;  /* 4-byte signed */
    typedef float   GLfloat;  /* single precision float */
    typedef float   GLclampf; /* single precision float in [0,1] */
    typedef double    GLdouble; /* double precision float */
    typedef double    GLclampd; /* double precision float in [0,1] */
```

Sources :

```
[1] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=823610
[2] https://www.khronos.org/opengl/wiki/OpenGL_Type
```

WatFiles

Sur le modèle de *WeshLoader* j'ai conçu *WatLoader*, capable d'écrire et de lire des matériaux au format Wat. Ce format commence par un header de 128 octets qui contient un ensemble de paramètres pour le modèle et la taille des textures s'il y en a. Le header comporte à l'instar de *WeshHeader* un magic number (0x4C544157 = ASCII(WATL)) et un numéro de version. Après le header on trouve des données uniformes (albédo, métallicité, rugosité et alpha). Puis les texture blocks (optionnels).

WatLoader se sert d'une structure *MaterialInfo* intermédiaire pour stocker les donnée sur le point d'être écrites ou qui viennent d'être lues. Cette structure permet à elle seule d'initialiser une *Texture* si tant est que des texture blocks existent. *WatLoader* peut aussi initialiser des *MaterialDescriptor*, essentiellement en se bornant à lire le header et les données uniformes d'un watfile. Ceci est utilisé par *MaterialFactory* pour chercher et stocker des données de base sur les matériaux des watfiles, comme elle le ferait pour des matériaux décrits par du XML.

Pour l'instant, il faut toujours déclarer les matérieaux dans assets.xml :

```
    <Materials>
        <Material location="beachSand.wat"/>
```

```
        <Material location="testMetalFloor01.wat"/>
        <Material location="testMetalFloor02.wat"/>
    </Materials>
```

Noter l'attribut *location* en lieu et place de *name*. C'est comme ça que le parser identifie qu'il s'agit d'un watfile. La chaîne contenue dans *location* sert cependant de nom pour l'identification ultérieure de l'asset. Ainsi on utilise toujours l'attribut *name* pour faire référence à un watfile, et on n'oublie pas le ".wat" :

```
    <Model>
        <Mesh type="cube"></Mesh>
        <Material name="erwinCube.wat"/>
    </Model>
```

Waterial a été modifié pour pouvoir exporter les matériaux sous ce format. Toutes les maps ont été modifiées pour se servir de watfiles là où c'est possible, et les watfiles sont naturellement compatibles avec les archives (pack0 en contient).

Améliorations possibles

[ ] Supporter les cubemaps [ ] Supporter les paramètres de textures suivants : [ ] Address UV (wrap/clamp) [ ] Min/Mag filter

[11-05-19]

Watfile support refactor

J'ai fait de grosses simplifications. L'ancienne structure *MaterialInfo* faisait pour l'essentiel doublon avec *MaterialDescriptor*. C'était voulu au départ, afin de forcer une séparation du traitement. J'ai viré *MaterialInfo* et augmenté *MaterialDescriptor* et *TextureDescriptor* afin d'y inclure les données essentielles au traitement des Watfiles. On peut maintenant utiliser la même structure pour gérer des Watfiles ou des matériaux XML. *MaterialFactory* est chargée de sélectionner le bon loader (*WatLoader* ou *PngLoader*) en fonction de l'origine des données, lors du chargement des textures. Elle alloue les buffers nécessaires et initialise des pointeurs de données dans le *TextureDescriptor*. *Texture* n'a que faire de l'origine des données et se sert simplement des pointeurs contenus dans le descripteur. *WatLoader* n'a plus de méthode read_descriptor(), juste une méthode read(3) qui prend en argument supplémentaire un booléen qui décide si les données de texture doivent être chargées ou non. L'idée est qu'on peut éviter de charger les données de texture tant qu'on n'en a pas besoin.

BUGS

[ ] In debug target [ ] *SoundSystem* fail FMOD assert (lib version != header version) [ ] Imgui fail assert (InitFrame() not called before Render())

[13-05-19]

En lisant un peu sur le batch rendering ([1] et [2]), après avoir un moment pesté contre mon incapacité à y avoir pensé plus tôt, j'ai remarqué que c'était déjà ce que je faisais dans les grandes lignes sans vraiment le savoir. Mes *BufferUnits* sont en réalité des render batches. J'ai donc renommé cette classe en *RenderBatch* et je compte l'équiper de quelques méthodes supplémentaires afin d'en assurer l'optimalité.

L'échange suivant entre Marek de marekknows.com et un autre gars dans [1] m'a fait assez sourire après avoir saisi la vraie nature de mes *BufferUnits* :

```
unbird> Nice article. One suggestion, though: I'd make the batch generic for
    arbitrary vertex types.

mmakrzem> Are you suggesting templating the Batch class to hold different types of
    vertices?  That's a good idea.

unbird> Precisely. :D
```

Yep. Les miens sont déjà templatés :)

[1] propose un système pour manager automatiquement des render batches. Chez-moi, cette tâche incombe au chunk system.

Sources :

```
[1] https://www.gamedev.net/articles/programming/graphics/opengl-batch
-rendering-r3900/
[2] https://gamedev.stackexchange.com/questions/65847/batching-elements
```

[20-05-19]

Grosse optimisation

Le G-Buffer et le L-Buffer partagent le même depth-buffer. Il s'ensuit que j'économise un blit lors de la light pass, et je gagne entre 2 et 3ms sur une frame.

Pour ce faire, j'ai modifié davantage le constructeur de *Texture* spécifique aux render targets pour prendre en premier argument une std::initializer_list. *TextureUnitInfo* regroupe les divers paramètres d'une texture unit et possède deux constructeurs. Le premier est publique et permet l'initialisation d'une texture unit depuis zéro. Le second est protégé (accessible depuis la friend class *Texture*) et génère un handle vers une texture unit déjà existante. La méthode Texture::share_unit(uint32_t index) permet de récupérer un tel handle depuis l'extérieur, qui servira à l'initialisation d'une autre texture :

```
    GMODULES::REGISTER(std::make_unique<BufferModule>
    (
        "gbuffer",
        std::make_unique<Texture>
        (
            std::initializer_list<TextureUnitInfo>
            {
                TextureUnitInfo("normalTex"_h, TextureFilter::MIN_NEAREST,
                   GL_RGBA16_SNORM,     GL_RGBA),
                TextureUnitInfo("albedoTex"_h, TextureFilter::MIN_NEAREST,
                   GL_RGBA,             GL_RGBA),
                TextureUnitInfo("depthTex"_h,  TextureFilter::MIN_NEAREST,
                   GL_DEPTH24_STENCIL8, GL_DEPTH_STENCIL)
            },
            GLB.WIN_W,
            GLB.WIN_H,
            TextureWrap::CLAMP_TO_EDGE
        ),
        std::vector<GLenum>({GL_COLOR_ATTACHMENT0,
                             GL_COLOR_ATTACHMENT1,
                             GL_DEPTH_STENCIL_ATTACHMENT})
    ));
```

```
21      GMODULES::REGISTER(std::make_unique<BufferModule>
        (
23          "lbuffer",
            std::make_unique<Texture>
25          (
                std::initializer_list<TextureUnitInfo>
27              {
                    TextureUnitInfo("screenTex"_h, TextureFilter::MIN_NEAREST,
                       GL_RGBA16F, GL_RGBA),
29                  TextureUnitInfo("brightTex"_h,
                       TextureFilter(TextureFilter::MAG_LINEAR |
                       TextureFilter::MIN_LINEAR_MIPMAP_LINEAR), GL_RGBA, GL_RGBA),
                    GMODULES::GET("gbuffer"_h).get_texture().share_unit(2)
31              },
                GLB.WIN_W,
33              GLB.WIN_H,
                TextureWrap::CLAMP_TO_EDGE,
35              true
            ),
37          std::vector<GLenum>({GL_COLOR_ATTACHMENT0,
                                 GL_COLOR_ATTACHMENT1,
39                               GL_DEPTH_STENCIL_ATTACHMENT})
        ));
```

Ici le L-Buffer récupère un tel handle sur la texture “depthTex” du G-Buffer, aucune nouvelle texture unit ne sera créée, le L-Buffer utilise effectivement le depth-buffer du G-Buffer.

[21-05-19]

GPU timer query

J’ai changé ma façon de mesurer les performances. J’y parviens maintenant sans recours à glFinish(), en utilisant une extension OpenGL timer query. La méthode adoptée n’introduit aucun pipeline stall et mesure effectivement le temps côté GPU. Les résultats des queries ne sont pas immédiatement disponibles, ainsi pour éviter de forcer une synchronisation je fais du query double buffering (voir [1] en fin de page) : je lance une “back query” avant le rendu, effectue le rendu, puis je stop la back query et lis le résultat de la “front query” avant de permuter les front/back query IDs.

```
        // * Initialization (just once)
2       unsigned int[2] query_ID_;
        unsigned int query_back_buffer_ = 0;
4       unsigned int query_front_buffer_ = 1;

6       glGenQueries(1, &query_ID_[query_back_buffer_]);
        glGenQueries(1, &query_ID_[query_front_buffer_]);
8
        // dummy query to prevent OpenGL errors from popping out during first frame
10      glBeginQuery(GL_TIME_ELAPSED, query_ID_[query_front_buffer_]);
        glEndQuery(GL_TIME_ELAPSED);
12
        // * Start
14      glBeginQuery(GL_TIME_ELAPSED, query_ID_[query_back_buffer_]);

16      // * Render stuff
        // glDrawMesCouilles()
```

```
18
    // * Stop
20  glEndQuery(GL_TIME_ELAPSED);
    glGetQueryObjectuiv(query_ID_[query_front_buffer_], GL_QUERY_RESULT,
        (GLuint*)&timer_);
22
    // * Swap query buffers
24  if(query_back_buffer_)
    {
26      query_back_buffer_ = 0;
        query_front_buffer_ = 1;
28  }
    else
30  {
        query_back_buffer_ = 1;
32      query_front_buffer_ = 0;
    }
```

La classe *GPUQueryTimer* encapsule ces fonctionnalités, et est utilisée par l'interface *Renderer* pour les mesures de performances :

```
1  void Renderer::Render(Scene* pscene)
   {
3      if(!enabled_)
           return;
5
       if(PROFILING_ACTIVE)
7          query_timer_.start();

9      render(pscene);

11     if(PROFILING_ACTIVE)
           dt_fifo_.push(query_timer_.stop());
13 }
```

La *RenderPipeline* va maintenant sommer les contributions de tous les renderers (non-debug) pour aboutir au draw time et n'utilise plus glFinish(). Le coût CPU pour la mesure de performances est drastiquement réduit. Les temps observés sont un poil plus faibles que précédemment.

Backface depth-buffer enabled SSR

La SSR utilise maintenant le backface depth buffer afin d'estimer l'épaisseur de la géométrie. Je constate une baisse significative des artéfacts liés aux faux positifs, au prix d'environ 1ms sur ma frame (ce que je peux maintenant me permettre).

La passe géométrique remplit un nouveau depth buffer (le GModule nommé *backfaceDepthBuffer*), mais cette fois en mode cull front. Ce depth-buffer contient donc l'information de profondeur des faces cachées de la géométrie. Cette information est utilisée à la place de l'offset constant (pixel thickness) dans la fonction d'évaluation d'intersection :

```
1  bool ray_intersects_depth_buffer(float rayZNear, float rayZFar, vec2 hitPixel)
   {
3      // Swap if bigger
       swap_if_bigger(rayZFar, rayZNear);
5      float cameraZ = -depth_view_from_tex(depthTex, hitPixel.xy,
           rd.v4_proj_params.zw);
       float backZ = -depth_view_from_tex(backDepthTex, hitPixel.xy,
           rd.v4_proj_params.zw);
```

```
    // Cross z
    return rayZFar <= cameraZ && rayZNear >= backZ;
}
```

BEWARE Une conséquence de l'utilisation de cette technique est que les réflexions disparaissent si la caméra se trouve à l'intérieur d'un objet de la scène : dans ce cas, la profondeur backface est toujours plus faible que la profondeur frontface, et le test d'intersection échoue systématiquement.

Sources :

```
[1] http://www.lighthouse3d.com/tutorials/opengl-timer-query/
[2] https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_timer_query.txt
```

[25-05-19]

[failed] Bloom pass optimization

J'ai essayé de réécrire la bloom pass pour éliminer la texture *brightTex*, en condensant l'information du bright mask dans le canal alpha de *screenTex*, afin d'économiser un multiple render target dans la light pass. En pratique cela me force à implémenter les bloom buffers en RGBA16F ce qui s'avère plus lent (en tout cas sur ma config).

[27-05-19]

Multi-line string

```
    std::string shader_src = R"(
        #version 330 core

        uniform float f_ma_couille;
        // ...
    )";
```

[03-06-19]

Git

Remove a git submodule

To remove a submodule you need to: * Delete the relevant section from the .gitmodules file. * Stage the .gitmodules changes git add .gitmodules * Delete the relevant section from .git/config. * Run git rm –cached path_to_submodule (no trailing slash). * Run rm -rf .git/modules/path_to_submodule (no trailing slash). * Commit git commit -m “Removed submodule” * Delete the now untracked submodule files rm -rf path_to_submodule

Update all submodules to latest commit from their remote

```
>> git submodule foreach git pull origin master
```

[04-06-19]

J'ai changé les formats internes des textures de rendu de la SSAO de GL_R8 vers GL_RGBA8, lors du refactor pour l'abstraction de l'API graphique. L'utilisation excessive des ressources GPU par la SSAO que je constatais

près des surfaces a disparu. J'ai maintenant un temps de rendu sensiblement égal quelle que soit ma position sur la map. Noter que sur le matos de Jess je n'ai jamais constaté ce problème. Il était donné d'avance qu'il pourrait y avoir un côut important en performances lié à l'utilisation de tels formats sur certaines machines, on recommande en général d'utiliser systématiquement un format à 4 canaux (RGBA). En tout cas je ne m'attendais pas à ce genre de profil de ralentissement, très ressemblant à du GPU cache thrashing.

SSAO mega plus rapide après avoir changé le format interne de GL_R8 à GL_RGBA8.

[10-06-19]

Je prototype un renderer multi-threaded, nom de code WCore2 (manque d'imagination). C'est un projet distinct de celui-ci, avec une prise de notes séparée. Je merge si mes expérimentations sont concluantes.

Vertex Buffer Layout

TheCherno a encore frappé, avec une chouette méthode pour abstraire les formats de vertex. Je me suis largement inspiré de son travail pour opérer un petit refactor de la fonction OGLVertexArray::set_layout(). Récemment j'étais déjà repassé sur cette fonctionnalité pour décider d'un layout dynamiquement en fonction du hash compile type (lib ctti) du type de vertex, l'implémentation était peu satisfaisante (spaghetti).

Le header vertex_format.h définit plusieurs nouvelles entités qui permettent la description de n'importe quel format de vertex, de manière très expressive. Le type énuméré *ShaderDataType* représente virtuellement tous les types de données que l'on peut passer à un shader, en particulier celui des attributs. La classe *BufferLayout* représente un layout, et consiste en un conteneur de *BufferLayoutElement*. Chaque élément représente un attribut par son nom, son type de données, sa taille en bytes, son offset dans la structure de vertex, plus un booléen pour la normalisation. *BufferLayout* est constructible depuis une initializer_list, ce qui permet une forme très expressive d'initialisation :

```
BufferLayout Vertex3P3N3T2U::Layout =
{
    {"a_position"_h, ShaderDataType::Vec3},
    {"a_normal"_h,   ShaderDataType::Vec3},
    {"a_tangent"_h,  ShaderDataType::Vec3},
    {"a_texCoord"_h, ShaderDataType::Vec2}
};

BufferLayout VertexAnim::Layout =
{
    {"a_position"_h, ShaderDataType::Vec3},
    {"a_normal"_h,   ShaderDataType::Vec3},
    {"a_tangent"_h,  ShaderDataType::Vec3},
    {"a_texCoord"_h, ShaderDataType::Vec2},
    {"a_weights"_h,  ShaderDataType::Vec4},
    {"a_boneIDs"_h,  ShaderDataType::IVec4},
};
```

Les offsets/sizes/strides de chaque élément sont calculés automatiquement à la construction du layout.

J'ai doté chaque structure de vertex d'un membre statique *BufferLayout*. Lors de l'initialisation d'un VAO il est alors aisé de faire :

```
VAO_ = VertexArray::create();
VAO_->bind();
VAO_->set_layout(VertexT::Layout);
```

Avec VertexT la structure de vertex.

BufferLayout est un type itérable. L'implémentation de set_layout pour OpenGL ressemble à ceci:

```
void OGLVertexArray::set_layout(const BufferLayout& layout) const
{
    uint32_t index = 0;
    for(const auto& element: layout)
    {
        glEnableVertexAttribArray(index);
        glVertexAttribPointer(index,
                              element.get_component_count(),
                              shader_data_type_to_ogl_base_type(element.type),
                              element.normalized ? GL_TRUE : GL_FALSE,
                              layout.get_stride(),
                              (const void*)(uint64_t)element.offset);
        ++index;
    }
}
```

Une simple boucle sur les éléments du layout, et une déduction des paramètres à fournir à l'API selon leur contenu. Elégant.

A noter que sous DirectX, le vertex buffer layout est lié au shader (pas de VAO sous DX), l'implémentation devrait changer de place un peu plus tard.

Elegant thread-safe singleton

Meyer's Singleton:

```
class Singleton
{
public:
    static Singleton& Instance()
    {
        static Singleton S;
        return S;
    }

private:
    Singleton();
    ~Singleton();
};
```

Scott Meyers says:

```
"This approach is founded on C++'s guarantee that local static objects are
   initialized when the object's definition is first encountered during a call to
   that function." ... "As a bonus, if you never call a function emulating a
   non-local static object, you never incur the cost of constructing and
   destructing the object."
```

-> thread-safe -> subject to Destruction Order Fiasco

TODO (Waterial): [X] Texmap controls dans une page dans un QTabWidget [ ] Faire une seconde page pour les propriétés générales [ ] Texture scale [ ] Options d'export [ ] FX ? [ ] Export des materials en XML [/] Choix des meshes (cube, plane, sphere, .obj) [/] Opérations de base sur chaque texture map [ ] Invert [ ] Bias [ ] Curve [X] HSV [ ] Export sous différentes résolutions

- TODO (WCore): [ ] New texture maps (possibly grouped in same Gbuffer chan): * Emissivity map * [.] Reflection map

[ ] Pre-multiplied alpha: https://www.essentialmath.com/GDC2015/VanVerth_Jim_DoingMathwRGB.pdf

[ ] Check extension support before using - GL_COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT (texture.cpp) - GL_COMPRESSED_RGBA_S3TC_DXT1_EXT (material_common.cpp) -> Il faut tester la présence de l'extension GL_EXT_texture_compression_s3tc. -> GLEW ne permet pas de détecter l'extension correctement, ne supporte pas vraiment les contextes core profile, et de plus nécessite glewExperimental pour ne pas segfault lamentablement lors d'un glGenVertexArrays(). **Passer sous GLAD** (penser à linker avec libdl sous nux (-ldl)). https://github.com/Dav1dde/glad https://glad.dav1d.de/ https://github.com/LibreVR/Revive/commit /86926af6908f7a99c443559a961b38b3ce33c74d

[ ] Perform texture compression offline. - Use glCompressedTexImage2D() https://opengl.developpez.com/tutoriels/opengl-tutorial /5-un-cube-texture/#LVII

[ ] Bien penser à updater les bounding boxes pour les objets qui bougent.

[ ] Insp better chunk loading:

```
What we do is actually pretty simple. Each frame we loop though all active chunks
    for update. During the update, we check and see if a chunk is missing any
    neighbors. If it is, we check and see if the neighbor chunk slots are withing
    the loading range. If they are, we load chunks and hook them up to their
    neighbors.
```

[ ] GL_RGBA -> GL_BGRA Il semblerait que le layout hardware le plus fréquent soit BGRA, et utiliser des framebuffers en RGBA force GL à swizzle. Donc GL_BGRA plus efficace en moyenne. Pour une conversion efficace des textures en BGRA on peut utiliser un swizzle mask :

```
GLint swizzleMask[] = {GL_BLUE, GL_GREEN, GL_RED, GL_ALPHA};
glTexParameteriv(GL_TEXTURE_2D, GL_TEXTURE_SWIZZLE_RGBA, swizzleMask);
```